# Az Android platform áttekintése

## 1. Mi az Android?

Az Android egy nyílt forráskódú, Linux-alapú operációs rendszer, amelyet elsősorban mobil eszközökre (okostelefonok, táblagépek) fejlesztettek ki. Az Android platformot a Google és az Open Handset Alliance fejleszti.

### Főbb jellemzők:

- Nyílt forráskódú
- Rugalmas és testreszabható
- Széles körben elterjedt (a mobil eszközök többsége Android-alapú)
- Gazdag fejlesztői ökoszisztéma

## 2. Android verziók és API szintek

Az Android rendszeresen kap frissítéseket, amelyek új funkciókat és fejlesztéseket hoznak. Minden Android verzióhoz tartozik egy API szint.

Néhány fontos mérföldkő:

- Android 4.4 (KitKat) - API 19
- Android 5.0 (Lollipop) - API 21
- Android 8.0 (Oreo) - API 26
- Android 10 - API 29
- Android 13 - API 33

Fejlesztőként fontos figyelembe venni a célzott Android verziót és a minimális támogatott API szintet.

## 3. Android alkalmazás komponensek

Az Android alkalmazások több komponensből épülhetnek fel:

1. **Activity**: Egy képernyő az alkalmazásban, amellyel a felhasználó interakcióba léphet.
2. **Service**: Háttérben futó komponens, amely hosszan futó műveleteket végezhet.
3. **Broadcast Receiver**: Rendszerszintű eseményekre reagáló komponens.
4. **Content Provider**: Alkalmazások közötti adatmegosztást tesz lehetővé.

## 4. Android alkalmazás életciklus

Az Android alkalmazások életciklusa kulcsfontosságú koncepció:

- onCreate()
- onStart()

- onResume()
- onPause()
- onStop()
- onDestroy()

Az alkalmazásoknak megfelelően kell kezelniük ezeket az életciklus eseményeket a hatékony erőforrás-kezelés és a jó felhasználói élmény érdekében.

## 5. Android fejlesztői eszközök

- **Android Studio**: A hivatalos integrált fejlesztői környezet (IDE) Android alkalmazások fejlesztéséhez.
- **Android SDK**: Szoftverfejlesztő készlet, amely tartalmazza a szükséges könyvtárakat és eszközöket.
- **Android Emulator**: Virtuális Android eszközök futtatására szolgáló eszköz.
- **ADB (Android Debug Bridge)**: Parancssori eszköz a fejlesztői eszköz és az Android eszköz közötti kommunikációhoz.

## 6. Android biztonság

Az Android több biztonsági mechanizmust is tartalmaz:

- Alkalmazás-szintű jogosultságok
- Sandboxing: minden alkalmazás saját virtuális környezetben fut
- Titkosítás
- Google Play Protect: malware elleni védelem

# Az Android Studio

## Android SDK beállítása

Az Android SDK (Software Development Kit) tartalmazza a fejlesztéshez szükséges eszközöket és API-kat.

1. Az Android Studio-ban navigáljon: Tools > SDK Manager
2. Az "SDK Platforms" fülön válassza ki a fejleszteni kívánt Android verziókat
3. Az "SDK Tools" fülön győződjön meg róla, hogy a következők be vannak jelölve:
    - Android SDK Build-Tools
    - Android Emulator
    - Android SDK Platform-Tools
    - Google Play services
4. Kattintson az "Apply" gombra a kiválasztott komponensek telepítéséhez

## Emulátor beállítása

Az emulátor lehetővé teszi Android eszközök szimulálását a számítógépen.

1. Navigáljon: Tools > AVD Manager
2. Kattintson a "Create Virtual Device" gombra
3. Válasszon egy eszköz definíciót (pl. Pixel 4)
4. Válasszon egy rendszerképet (lehetőleg x86 vagy x86_64 architektúrát az optimális teljesítményért)
5. Konfigurálja az emulátort (név, orientáció, stb.)
6. Kattintson a "Finish" gombra

## Gradle beállítása

A Gradle az Android projektek build rendszere. Az Android Studio automatikusan konfigurálj

a, de néha szükség lehet a beállítások módosítására.

1. Nyissa meg a projekt build.gradle fájlját
2. Ellenőrizze a Gradle verzióját és az Android Gradle Plugin verzióját
3. Szükség esetén módosítsa a distributionUrl-t a legfrissebb Gradle verzióra

## Egyéb hasznos beállítások

- Automatikus importálás engedélyezése: File > Settings > Editor > General > Auto Import
- Kódformázás beállítása: File > Settings > Editor > Code Style
- Élő sablonok testreszabása: File > Settings > Editor > Live Templates

# XML szintaxis és struktúra az Android fejlesztésben

Az XML (eXtensible Markup Language) kulcsfontosságú szerepet játszik az Android alkalmazások felhasználói felületeinek definiálásában. Ebben a részben áttekintjük az XML alapjait és annak Android-specifikus használatát.

# 1. XML alapok

### 1.1 XML dokumentum struktúra

Egy XML dokumentum általában a következő részekből áll:

1. XML deklaráció
2. Gyökérelem
3. Gyermekelemek
4. Attribútumok
5. Tartalmi szöveg

### 1.2 XML szintaxis szabályok

- Minden megnyitott tag-et be kell zárni
- A tag-ek kis- és nagybetű érzékenyek
- A tag-eket megfelelően kell egymásba ágyazni
- Az attribútum értékeket idézőjelek közé kell tenni

# 2. Android-specifikus XML struktúra

Az Android layoutok XML fájlokban vannak definiálva, általában a res/layout/ könyvtárban.

## 2.1 Gyökérelem

Android layoutokban a gyökérelem általában egy ViewGroup, például:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <!-- Gyermekelemek itt -->

</LinearLayout>
```

## 2.2 Névterek

Az Android XML fájlokban fontos a névterek használata. A leggyakoribb az android névtér:

```
xmlns:android="http://schemas.android.com/apk/res/android"
```

Ez lehetővé teszi az Android-specifikus attribútumok használatát, például:

```
android:layout_width="match_parent"
```

## 2.3 View elemek

A gyökérelemen belül különböző View elemeket definiálhatunk:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello, World!" />

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Click me" />
```

## 2.4 Attribútumok

Az attribútumok határozzák meg a View-k tulajdonságait:

- android:layout_width és android:layout_height: A View méretezése
- android:id: Egyedi azonosító a View-hoz
- android:text: A View-n megjelenő szöveg
- android:onClick: A kattintáskor végrehajtandó metódus neve

Példa:

```
<Button
    android:id="@+id/myButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_text"
    android:onClick="onButtonClick" />
```

### 2.5 Erőforrás hivatkozások

Az XML-ben hivatkozhatunk más erőforrásokra a @ szimbólum használatával:

- @string/: String erőforrások
- @drawable/: Képek és ikonok
- @color/: Színek
- @dimen/: Méretek

Példa:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_text"
    android:textColor="@color/primary_text"
    android:textSize="@dimen/large_text" />
```

# 3. Gyakori ViewGroup-ok

A Lineáris elrendezés (LinearLayout) megismerése és testreszabása, amely lehetővé teszi a komponensek egymás alá (vertikálisan) vagy mellé (horizontálisan) rendezését. Az óra végére a hallgatók képesek lesznek létrehozni és konfigurálni egy LinearLayout-alapú felületet, figyelembe véve a komponensek arányosítását és igazítását.

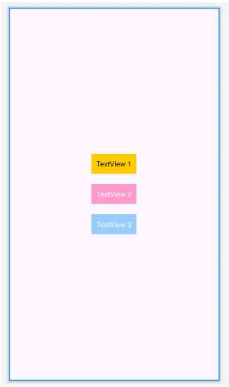
## LinearLayout

1. **Mi az a LinearLayout?**
    - Az Android felhasználói felületének egyik alapvető elrendezési módja, amely lehetővé teszi a komponensek egymás alá (vertikálisan) vagy mellé (horizontálisan) helyezését egy egyenes vonalban.
    - Két fő orientáció:
        - android:orientation="vertical" (egymás alá rendezés)

- android:orientation="horizontal" (mellé rendezés)

2. **LinearLayout tulajdonságai:**
   - **android:gravity**: Meghatározza a komponensek igazítását a LinearLayout területén belül (pl. középre igazítás, jobb vagy bal szélhez igazítás).
   - **android:layout_gravity**: Egy adott komponens igazítása a LinearLayout más komponenseihez képest (pl. egy gomb középre igazítása).
   - **android:layout_weight**: Lehetővé teszi az elemek arányos szélességű/magasságú elosztását a rendelkezésre álló hely szerint. Fontos a rendelkezésre álló hely dinamikus kiosztásához.
3. **Komponensek elhelyezése és igazítása:**
   - Hogyan helyezzük el a komponenseket a layout_width és layout_height attribútumok használatával: wrap_content (a tartalomhoz igazodik) és match_parent (kitölti a szülő nézetet).
   - padding és margin: különbségek és alkalmazásuk a komponensek között és a LinearLayout-ban.
4. **Az elrendezés optimalizálása:**
   - Hogyan oszthatjuk fel a rendelkezésre álló helyet a komponensek között a layout_weight segítségével. Példa: három gomb elhelyezése egyenlő szélességben egy sorban.
   - A LinearLayout használata többféle képernyőmérethez.

## Gyakorlat:

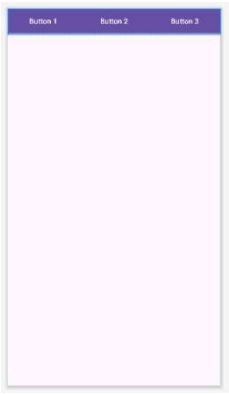



### 1. Feladat: Egyszerű LinearLayout létrehozása vertikális elrendezéssel

- A felhasználói felület tartalmazzon három TextView elemet, amelyeket egymás alá rendezünk.
- Minden TextView-hoz különböző színt állítunk be a háttérhez és a szöveghez.
- Minden TextView-nak legyen margója a széleitől és középen legyen elhelyezve.

```
<LinearLayout  xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="TextView 1"
        android:background="#FFCC00"
        android:textColor="#000000"
```

```
        android:padding="10dp"
        android:layout_marginBottom="20dp"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="TextView 2"
        android:background="#FF99CC"
        android:textColor="#FFFFFF"
        android:padding="10dp"
        android:layout_marginBottom="20dp"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="TextView 3"
        android:background="#99CCFF"
        android:textColor="#FFFFFF"
        android:padding="10dp"/>

</LinearLayout>
```

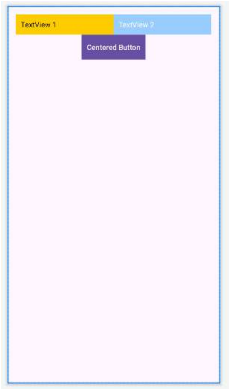



### 2. Feladat: Horizontális LinearLayout létrehozása súlyozással

- Három gomb (Button) egymás mellett helyezkedjen el egy sorban.
- A gombok egyenlő szélességűek legyenek a layout_weight használatával.
- Minden gombhoz különböző háttérszínt rendelünk.

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <Button
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Button 1"
        android:layout_weight="1"
```

```
        android:background="#FF0000"
        android:textColor="#FFFFFF" />

    <Button
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Button 2"
        android:layout_weight="1"
        android:background="#00FF00"
        android:textColor="#FFFFFF" />

    <Button
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Button 3"
        android:layout_weight="1"
        android:background="#0000FF"
        android:textColor="#FFFFFF" />

</LinearLayout>
```




### 3. Feladat: Kombinált LinearLayout vertikális és horizontális elrendezésekkel

- Készíts egy összetett elrendezést, ahol a fő LinearLayout vertikális elrendezésű.
- A fő layout tartalmazzon két sor elemet:
    - Az első sorban két TextView legyen egymás mellett, egyenlő szélességben.
    - A második sorban egy Button legyen középre igazítva.

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">
```

```
        <TextView
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:text="TextView 1"
            android:layout_weight="1"
            android:background="#FFCC00"
            android:textColor="#000000"
            android:padding="10dp"/>

        <TextView
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:text="TextView 2"
            android:layout_weight="1"
            android:background="#99CCFF"
            android:textColor="#FFFFFF"
            android:padding="10dp"/>

    </LinearLayout>

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Centered Button"
        android:layout_gravity="center"
        android:background="#FF99CC"
        android:textColor="#FFFFFF"
        android:padding="10dp" />

</LinearLayout>
```

# RelativeLayout

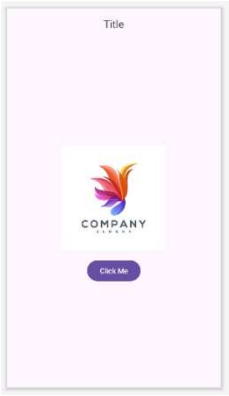
1. **Mi az a RelativeLayout?**
    - A RelativeLayout egy olyan elrendezés, ahol az egyes komponensek helyzete más komponensekhez vagy a szülő nézethez viszonyítva van megadva.
    - Lehetővé teszi a komponensek elhelyezését egymáshoz képest: pl. „egy gomb a szöveg alatt”, „egy kép a szülő nézet jobb oldalán”.
2. **RelativeLayout tulajdonságai:**
    - **android:layout_alignParentTop**, **android:layout_alignParentBottom**, **android:layout_alignParentLeft**, **android:layout_alignParentRight**: Az elem elhelyezése a szülő nézet valamelyik széléhez igazítva.
    - **android:layout_below**, **android:layout_above**: Egy komponens elhelyezése egy másik komponens alatt vagy fölött.
    - **android:layout_toRightOf**, **android:layout_toLeftOf**: Egy komponens elhelyezése egy másik komponens jobbra vagy balra eső oldalára.

- **android:layout_centerInParent**, **android:layout_centerHorizontal**, **android:layout_centerVertical**: Az elem középre igazítása a szülő nézeten belül.

3. **Komponensek egymáshoz viszonyított elhelyezése:**
   - A komponensek relatív pozicionálása egy másik elemhez, például egy gomb elhelyezése egy képfelirat alatt.
   - Hogyan helyezzük el a komponenseket úgy, hogy azok rugalmasan alkalmazkodjanak különböző képernyőméretekhez.
4. **Komponensek igazítása és összetett elrendezések:**
   - A layout_margin és a layout_padding szerepe a komponensek távolságának meghatározásában.
   - Összetett elrendezések készítése több komponens elhelyezésével egymás fölé, mellé vagy középre.

## Gyakorlat:

### 1. Feladat: Alapvető RelativeLayout használata




- A feladat során egy RelativeLayout alapú felületet kell létrehozni, amely tartalmaz egy TextView-t a szülő nézet tetején, egy képet (ImageView) a szülő nézet közepén, és egy gombot (Button), amely a kép alatt helyezkedik el.

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/titleTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Title"
        android:textSize="20sp"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:padding="16dp" />

    <ImageView
        android:id="@+id/centerImage"
        android:layout_width="200dp"
        android:layout_height="200dp"
        android:src="@drawable/sample_image"
        android:layout_centerInParent="true" />

    <Button
        android:id="@+id/belowButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me"
```

```
        android:layout_below="@id/centerImage"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="16dp" />

</RelativeLayout>
```

### 2. Feladat: Több elem egymáshoz viszonyított elhelyezése

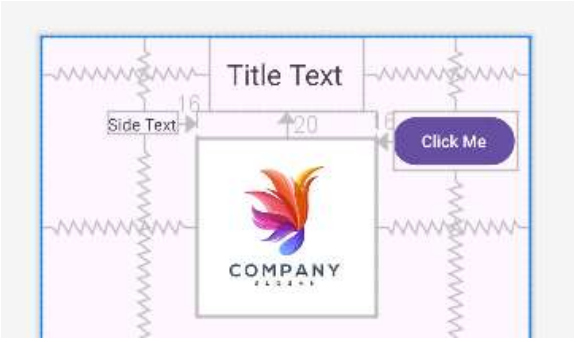

- Készíts egy olyan elrendezést, ahol két TextView van egymás mellett, de az egyik a másik jobb oldalán helyezkedik el. A gombot helyezd a képernyő jobb alsó sarkába.

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/leftTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Left Text"
        android:layout_alignParentLeft="true"
        android:layout_centerVertical="true" />

    <TextView
        android:id="@+id/rightTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Right Text"
        android:layout_toRightOf="@id/leftTextView"
        android:layout_marginLeft="16dp"
        android:layout_centerVertical="true" />

    <Button
        android:id="@+id/rightBottomButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Right Bottom"
        android:layout_alignParentBottom="true"
        android:layout_alignParentRight="true"
        android:layout_margin="16dp" />

</RelativeLayout>
```

### 3. Feladat: Bonyolultabb elrendezés több komponenssel


Hozz létre egy felületet, ahol:

- Egy cím (TextView) van a képernyő tetején, középre igazítva.
- Alatta egy kép (ImageView), amely középen van elhelyezve.
- A kép bal oldalán egy másik TextView, és a jobb oldalán egy gomb.




```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Title Text"
        android:textSize="24sp"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:padding="16dp" />

    <ImageView
        android:id="@+id/mainImage"
        android:layout_width="150dp"
        android:layout_height="150dp"
        android:layout_below="@id/title"
        android:layout_centerHorizontal="true"
        android:src="@drawable/sample_image"
        android:layout_marginTop="20dp" />

    <TextView
        android:id="@+id/sideText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Side Text"
        android:layout_toLeftOf="@id/mainImage"
        android:layout_below="@id/title"
        android:layout_marginRight="16dp"
        android:layout_centerVertical="true" />
```

```
    <Button
        android:id="@+id/sideButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me"
        android:layout_toRightOf="@id/mainImage"
        android:layout_below="@id/title"
        android:layout_marginLeft="16dp"
        android:layout_centerVertical="true" />

</RelativeLayout>
```

# ConstraintLayout

1. **Mi az a ConstraintLayout?**
   - A ConstraintLayout egy olyan elrendezés, amely lehetővé teszi a komponensek egymáshoz és a szülő nézethez való rögzítését.
   - Előnye, hogy egyetlen elrendezési struktúrán belül létrehozhatunk komplex layoutokat, amit korábban több különböző elrendezéssel (LinearLayout, RelativeLayout) kellett megoldani.
   - Jobb teljesítményt biztosít, mivel kevesebb nested layoutra van szükség, így csökkenti a hierarchia mélységét.
2. **Működése:**
   - A ConstraintLayout lényege, hogy a komponensek pozícióját és méretét úgy határozzuk meg, hogy korlátokat (constraints) rendelünk hozzájuk.
   - Egy elem legalább két korláttal kell rendelkezzen: vízszintes (horizontal) és függőleges (vertical).
   - Korlátok típusai:
     - **A szülő nézethez való rögzítés**: például egy elem rögzíthető a szülő nézet bal, jobb, felső vagy alsó széléhez.
     - **Egy másik elemhez való rögzítés**: például egy elem elhelyezése egy másik elem bal oldalán, felett, alatt, stb.
     - **Baseline rögzítés**: két elem szöveg alapjának összekapcsolása, hogy azonos magasságban legyenek.
3. **Alapvető ConstraintLayout attribútumok:**
   - **layout_constraintLeft_toLeftOf**, **layout_constraintRight_toRightOf**, **layout_constraintTop_toTopOf**, **layout_constraintBottom_toBottomOf**: Ezekkel rögzítjük a komponenseket a szülőhöz vagy más elemekhez.
   - **Chain-ek** (láncok) használata: Ha több elemet szeretnénk egymáshoz igazítani és rugalmasan elosztani a rendelkezésre álló területen, láncokat használhatunk. A láncokban az elemek közötti hely automatikusan kiegyensúlyozható.
   - **Bias** használata: A bias lehetővé teszi, hogy egy elem helyzetét rugalmasan módosítsuk a szülőhöz viszonyítva, például a középponttól balra vagy jobbra csúsztassuk azt.
4. **Méretmeghatározás ConstraintLayout-ban:**
   - **match_constraint** (0dp): Az elem kitölti a rendelkezésre álló helyet a korlátok alapján.
   - **wrap_content**: Az elem mérete a tartalomhoz igazodik.
   - **Fixed size**: Az elem konkrét szélességgel és magassággal rendelkezik.
5. **ConstraintLayout fejlettebb funkciói:**
   - **Guideline**: Ezek virtuális vonalak, amelyeket arra használhatunk, hogy az elemeket vízszintesen vagy függőlegesen igazítsuk bizonyos százalékos vagy konkrét értékű helyzetekhez.
   - **Barrier**: Barriereket használhatunk arra, hogy egy elem több másik elem változó méretéhez igazodjon dinamikusan.
   - **Group**: Egy csoporthoz több elemet is hozzárendelhetünk, amelyeket aztán egyszerre lehet mutatni vagy elrejteni.

*Gyakorlat:*

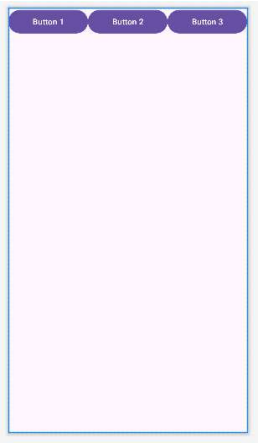
## 1. Feladat: Egyszerű ConstraintLayout felület létrehozása


- Hozz létre egy felületet, amely tartalmaz egy címkét (TextView), egy képet (ImageView) és egy gombot (Button).
- A címkét a képernyő tetejéhez igazítsd, a képet a közepére, és a gombot a képernyő aljára.
- A gomb legyen középre igazítva, míg a címke és a kép legyen a képernyő szélességéhez igazítva.

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <!-- Címke (TextView) -->
    <TextView
        android:id="@+id/titleText"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Title"
        android:textSize="24sp"
        android:textAlignment="center"
        android:layout_marginTop="16dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent" />

    <!-- Kép (ImageView) -->
    <ImageView
        android:id="@+id/imageView"
        android:layout_width="200dp"
        android:layout_height="200dp"
        android:src="@drawable/sample_image"
        app:layout_constraintTop_toBottomOf="@id/titleText"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"/>

    <!-- Gomb (Button) -->
    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        android:layout_marginBottom="16dp" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

---

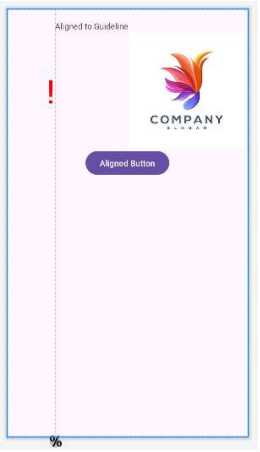
## 2. Feladat: Képernyő három gombbal




- Hozz létre egy felületet három gombbal, amelyek egy sorban helyezkednek el, egyenlő arányban osztva el a rendelkezésre álló területet.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <!-- Gomb 1 -->
    <Button
        android:id="@+id/button1"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Button 1"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toLeftOf="@id/button2"
        app:layout_constraintTop_toTopOf="parent"
       />

    <!-- Gomb 2 -->
    <Button
        android:id="@+id/button2"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Button 2"
        app:layout_constraintLeft_toRightOf="@id/button1"
        app:layout_constraintRight_toLeftOf="@id/button3"
        app:layout_constraintTop_toTopOf="parent" />

    <!-- Gomb 3 -->
    <Button
        android:id="@+id/button3"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Button 3"
        app:layout_constraintLeft_toRightOf="@id/button2"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

### 3. Feladat: Képernyő GuideLine-nal




Készíts egy felhasználói felületet **ConstraintLayout** használatával, amely tartalmaz:

1. Guideline (irányvonal), amely a képernyő szélességének 20%-ánál helyezkedik el.
2. Egy TextView címkét, amely a Guideline-hoz van igazítva.
3. Egy ImageView képet, amely a címke jobb oldalán és alatta helyezkedik el.
4. Egy Button gombot, amely a kép alatt középen található.
5. Egy piros színű, nagy méretű TextView-t ("!"), amely a Guideline-tól balra, a képpel egy vonalban helyezkedik el.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <!-- Irányvonal (Guideline) a képernyő 20%-ánál -->
    <androidx.constraintlayout.widget.Guideline
        android:id="@+id/guideline"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        app:layout_constraintGuide_percent="0.2" />

    <!-- Címke az irányvonalhoz igazítva -->
    <TextView
        android:id="@+id/title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Aligned to Guideline"
        app:layout_constraintLeft_toLeftOf="@id/guideline"
        app:layout_constraintTop_toTopOf="parent"
        android:layout_marginTop="20dp" />

    <!-- Kép a barrierhez igazítva -->
    <ImageView
        android:id="@+id/image"
        android:layout_width="200dp"
        android:layout_height="200dp"
        android:src="@drawable/logo2"
        app:layout_constraintLeft_toRightOf="@id/title"
        app:layout_constraintTop_toBottomOf="@id/title" />

    <!-- Gomb -->
    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Aligned Button"
        app:layout_constraintTop_toBottomOf="@id/image"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"/>
```

```
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="!"
            android:textSize="56sp"
            android:textColor="#ff0000"
            app:layout_constraintEnd_toStartOf="@id/guideline"
            app:layout_constraintTop_toTopOf="@id/image"
            app:layout_constraintBottom_toBottomOf="@id/image"/>

    </androidx.constraintlayout.widget.ConstraintLayout>
```

# Gyakorlati feladat (Időjárás)

## Adatmodell (WeatherResponse) létrehozása

Létre kell hoznunk egy adatmodellt, amely megfelel az OpenWeatherMap válaszának. Mivel először csak a hőmérsékletet akarjuk megjeleníteni, csak azt helyezzük el az osztályban az adatszerkezetből.

**WeatherResponse.kt**

```
data class WeatherResponse(
    val main: Main,
)

data class Main(
    val temp: Float
)
```

**Magyarázat**

- A `WeatherResponse` adatmodell a JSON válasz struktúráját tükrözi. Ez tartalmazza a `main` mezőt, amely az időjárási adatokat tartalmazza (pl. hőmérséklet és páratartalom), de ebből csak a hőmérséklet kell.

## Szükséges libraryk, a Retrofit és Gson hozzáadása

### A `lib.versions.toml` frissítése

Menj a projekted `gradle/libs.versions.toml` fájljába, és add hozzá a Retrofit és a Gson könyvtárak verzióit és definícióit. A fájl struktúrája így nézhet ki:

```
[versions]
retrofit = "2.9.0"
gson = "2.9.0"
okhttp = "4.9.3"

[libraries]
retrofit = { module = "com.squareup.retrofit2:retrofit", version.ref =
"retrofit" }
retrofit-gson = { module = "com.squareup.retrofit2:converter-gson",
version.ref = "gson" }
okhttp = { module = "com.squareup.okhttp3:okhttp", version.ref = "okhttp" }
```

**Magyarázat:**

- **[versions]**: Itt definiálod az egyes könyvtárak verzióját. Ebben az esetben `retrofit` és `gson` verziókat adsz meg.
- **[libraries]**: Itt definiálod a könyvtárakat a modul és a verzió hivatkozásával. A `version.ref` segítségével hivatkozunk a fent megadott verziókra.
- Az `okhttp`-ot is hozzáadtam, mert a Retrofit az **OkHttp** alapjaira épül, és ezt esetlegesen felhasználhatjuk az extra funkciókhoz, például a naplózáshoz.

### Az `app/build.gradle` frissítése

Miután a `libs.versions.toml` fájlt frissítetted, nyisd meg az `app/build.gradle` fájlt, és add hozzá a szükséges könyvtárakat a függőségek (`dependencies`) részhez:

```
dependencies {
    implementation(libs.retrofit)
    implementation(libs.retrofit.gson)
    implementation(libs.okhttp)
}
```

**Magyarázat:**

- **libs.retrofit** és **libs.retrofit.gson**: Ezeket a `lib.versions.toml`-ban definiáltuk. Az `implementation(libs.retrofit)` parancs segít a Retrofit hozzáadásában a projekthez.
- Az `okhttp` szintén szükséges lehet, különösen ha saját interceptorokat akarsz beállítani vagy finomhangolni szeretnéd a hálózati kéréseket.

## Összefoglalás

- **libs.versions.toml**: Itt definiáltuk a verziókat és a megfelelő könyvtárakat.
- **app/build.gradle**: Itt adtuk hozzá a megfelelő hivatkozásokat a `dependencies` blokkhoz.

Ez segít majd abban, hogy a függőségeid egy helyen legyenek kezelve, és könnyebben frissíthesd azokat, ha szükséges.

## Internet engedély megadása

Manifest fájlban adjuk meg, néküle nincs internetes kommunikáció:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

## WeatherService interfész létrehozása

Hozzuk létre a `WeatherService` interfészt, amely meghatározza, hogyan kell kinéznie az API hívásoknak.

**WeatherService.kt**

```
interface WeatherService {
    @GET("data/2.5/weather")
    fun getWeather(
        @Query("q") cityName: String,
        @Query("appid") apiKey: String,
        @Query("units") units: String
```

```
    ): Call<WeatherResponse>
}
```

**Magyarázat**

- **@GET** annotáció: A végpont (`/data/2.5/weather`) megadásával lekérjük az időjárási adatokat.
- **@Query**: Paramétereket adunk meg, például a város nevét (`q`), az API kulcsot (`appid`), és a mérési egységet (`units` - ahol "metric" a Celsius).

## A `fetchWeatherData()` magyarázata

Ez a metódus felelős az időjárási adatok lekéréséért a OpenWeatherMap API használatával. Menjünk végig a metódus minden során, és értsük meg, hogyan működik a hálózati kérés, a válasz feldolgozása, valamint a UI frissítése.

```
1. val retrofit = Retrofit.Builder()...
val retrofit = Retrofit.Builder()
    .baseUrl("https://api.openweathermap.org/")
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

- **Retrofit példány létrehozása**: Ez a rész a Retrofit objektum létrehozásával kezdődik, amely egy HTTP kliens a REST API-khoz.
- **baseUrl()**: Meghatározzuk az alap URL-t (baseUrl), amely az összes API végpont közös része. Ebben az esetben az OpenWeatherMap alapcíme van megadva: "https://api.openweathermap.org/".
- **addConverterFactory()**: A GsonConverterFactory hozzáadásával tudjuk a JSON válaszokat Kotlin objektumokká alakítani. Ez automatikusan megtörténik, miután a Retrofit válasz visszaérkezett. Itt a **Gson** segítségével konvertáljuk a JSON formátumú választ.
- **build()**: Ezzel létrehozzuk a Retrofit objektumot a meghatározott konfigurációkkal.

```
2. val weatherService = retrofit.create(WeatherService::class.java)
val weatherService = retrofit.create(WeatherService::class.java)
```

- **API interfész implementálása**: A Retrofit segítségével létrehozunk egy `WeatherService` példányt. A `WeatherService` egy **interfész**, amely meghatározza, hogyan néz ki az API hívás.
- **retrofit.create()**: Ezzel a módszerrel a Retrofit létrehoz egy implementációt a `WeatherService` interfészhez, így használhatjuk azt a hálózati kérés indításához.

```
3. val call = weatherService.getWeather("Tatabánya", apiKey, "metric")
val call = weatherService.getWeather("Tatabánya", apiKey, "metric")
```

- **API hívás előkészítése**: Meghívjuk a `getWeather` metódust a `weatherService` példányon keresztül. Ez a metódus előre meghatározott az interfészben.
    - **"Tatabánya"**: A város neve, amiről az időjárási adatokat kérjük le.
    - **apiKey**: Az API kulcs, amit az OpenWeatherMap-től kaptál.
    - **"metric"**: A mértékegység típusa, ebben az esetben Celsius fokban szeretnénk megkapni a hőmérsékletet (`"metric"`).

Ez a hívás egy `Call<WeatherResponse>` objektumot hoz létre, amely majd tartalmazza az API válaszát.

*4. `call.enqueue(...)`*

```
call.enqueue(object : Callback<WeatherResponse> {
    override fun onResponse(
        call: Call<WeatherResponse>,
        response: Response<WeatherResponse>
    ) {
        if (response.isSuccessful) {
            val weatherResponse = response.body()
            if (weatherResponse != null) {
                val weatherInfo = weatherResponse.main.temp

                // Eredmény kiírása a logba
                Log.d("WeatherActivity", weatherInfo.toString())

                // Eredmény kiírása a képernyőre
                textviewTemp.text = weatherInfo.toString()
            }
        }
    }

    override fun onFailure(call: Call<WeatherResponse>, t: Throwable) {
        Log.e("WeatherActivity", "Hiba az API lekérés során", t)
    }
})
```

- **Aszinkron API hívás (`enqueue()`)**: A `call.enqueue()` aszinkron módon indítja el a hálózati kérést. Ez azt jelenti, hogy a hálózati kérés és válasz fogadása nem blokkolja az alkalmazás főszálát. Két fő függvényt definiálunk benne:
    1. **onResponse()**: Akkor hívódik meg, ha a válasz sikeresen megérkezett az API-tól.
    2. **onFailure()**: Akkor hívódik meg, ha hiba történik a hálózati kérés során (például nincs internetkapcsolat vagy az API nem elérhető).

`onResponse()`

- **`if (response.isSuccessful)`**:
    - Ellenőrizzük, hogy a válasz sikeres volt-e. A `response.isSuccessful` igaz, ha az HTTP státusz kód 200-as (vagyis sikeres).
- **`val weatherResponse = response.body()`**:
    - Ha a válasz sikeres, megszerezzük a válasz bodyt, ami egy `WeatherResponse` típusú objektum. Ez az objektum tartalmazza a letöltött időjárási adatokat.
- **`if (weatherResponse != null)`**:
    - Ha a válasz nem üres (`null`), folytatjuk az adat feldolgozásával.
- **`val weatherInfo = weatherResponse.main.temp`**:
    - Kinyerjük a hőmérséklet adatot (`temp`) a `main` objektumból.
- **Log kiírás**:
    - Az adatot kiírjuk a logba, hogy lássuk az időjárási adatokat (`Log.d("WeatherActivity", weatherInfo.toString())`).
- **UI frissítése**:
    - Az `textviewTemp` nevű `TextView` UI elemet frissítjük az időjárási adat (`temp`) megjelenítésével (`textviewTemp.text = weatherInfo.toString()`).

`onFailure()`

- **`Log.e("WeatherActivity", "Hiba az API lekérés során", t)`**:
    - Ha a kérés sikertelen, akkor kiírjuk a hibát a logba. Ez segít hibakeresésben és annak megértésében, hogy miért nem sikerült a kérés (pl. hálózati hiba, rossz API kulcs).

## Összefoglalás

1. **Retrofit Példány Létrehozása**:
   - Létrehozzuk a Retrofit példányt a megadott alap URL-lel és egy `GsonConverterFactory`-val, ami segít a JSON feldolgozásában.
2. **WeatherService API Interfész Létrehozása**:
   - A Retrofit segítségével létrehozunk egy implementációt az API interfészhez (`WeatherService`).
3. **API Hívás Meghívása**:
   - Az `getWeather()` metódussal elkészítjük az API hívást, amely `Tatabánya` időjárási adatait kéri le.
4. **Kérés Indítása Aszinkron Módon (`enqueue()`)**:
   - Az API hívást aszinkron módon indítjuk el, és kezeljük a sikeres választ (`onResponse()`) vagy a hibát (`onFailure()`).
   - A sikeres válasz esetén kinyerjük a hőmérsékletet, és azt kiírjuk a logba, majd megjelenítjük az alkalmazás UI-án is.

---

# A RecyclerView

## Mi az a RecyclerView, és miért használjuk?

Az **RecyclerView** az Android fejlesztésben az adatok hatékony megjelenítésére szolgál, például listák vagy rácsszerkezetek formájában.

1. **View Recycling (Újrafelhasználás):** Csak azokat az elemeket tölti be, amelyek éppen a képernyőn láthatók, ezzel memóriát takarít meg.
2. **LayoutManager:** Meghatározza az elemek elrendezését (például lineáris lista, rács, stb.).
3. **Adapter:** Az adatok kezelése és az elemek feltöltése az egyéni nézetekbe.
4. **ViewHolder Pattern:** Az elemek gyorsabb megjelenítésére szolgál azáltal, hogy a nézeteket gyorsítótárazza.

## RecyclerView létrehozása

## 1. Felhasználói felület

Az XML fájlban az alábbi kóddal definiáljuk a RecyclerView-t, ez önmagában jelenti az egész görgethető listát:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="8dp"
    android:background="@android:color/white"/>
```

## 2. Adatmodell létrehozása (User.kt)

Az adatmodell azokat az adatok reprezentálja, amelyeket meg akarunk jeleníteni. Ebből általában egy listánk lesz, ezt a listát tartalmazza a RecyclerView. Itt létrehozunk egy `User` osztályt, amely a következő adatokat tartalmazza:

```
data class User(
    val name: String,
    val email: String,
    val profileImage: Int // Drawable resource ID
)
```

### 3. Egyéni listaelem létrehozása (list_item.xml)

A listában levő elemek kinézetét ebben az XML fájlban definiáljuk, tehát ez csak 1 user megjelenítését határozza meg egy önálló xml fájl formájában:

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="12dp"
    android:orientation="horizontal">

    <ImageView
        android:id="@+id/profileImage"
        android:layout_width="50dp"
        android:layout_height="50dp"
        android:src="@drawable/ic_user"/>

    <LinearLayout
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:orientation="vertical">

        <TextView
            android:id="@+id/nameText"
            android:textSize="18sp"
            android:textStyle="bold"/>

        <TextView
            android:id="@+id/emailText"
            android:textSize="14sp"
            android:textColor="@android:color/darker_gray"/>
    </LinearLayout>

</LinearLayout>
```

### 4. Adapter létrehozása (UserAdapter.kt)

# 1. Az Adapter osztály és a generikus típusok

```
class UserAdapter(private val userList: List<User>) :
RecyclerView.Adapter<UserAdapter.UserViewHolder>()
```

- **`RecyclerView.Adapter<UserAdapter.UserViewHolder>()`**
  Ez az adapter osztály az `RecyclerView.Adapter`-ből származik, és egyedi `UserViewHolder` típusát használja.
  Az adapter felelős az egyes nézetek létrehozásáért és az adatok megjelenítéséért.

- **`private val userList: List<User>`**
  Az adapter konstruktorában átadunk egy **userList** nevű listát, amely az adatok gyűjteménye. A lista elemeit meg kell jeleníteni az egyes listanézetekben.

# 2. ViewHolder osztály

```
class UserViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
    val profileImage: ImageView = itemView.findViewById(R.id.profileImage)
    val nameText: TextView = itemView.findViewById(R.id.nameText)
    val emailText: TextView = itemView.findViewById(R.id.emailText)
}
```

### Miért van szükség ViewHolder-re?

- A ViewHolder koncepció segít **optimalizálni a lista teljesítményét**, elkerülve a felesleges hívásokat a `findViewById()` metódusra.
- Az újrafelhasználás miatt a nézeteket gyorsabban tudjuk be- és kirenderelni.
- **`RecyclerView.ViewHolder(itemView)`**: Az alap ViewHolder osztályt örökli, amely tartalmazza az adott listaelem nézetének referenciaobjektumát.

### Mezők:

- **`profileImage: ImageView`** – A profilkép megjelenítésére szolgáló képnézet.
- **`nameText: TextView`** – A felhasználó nevének megjelenítésére szolgáló szöveges nézet.
- **`emailText: TextView`** – Az e-mail cím megjelenítésére szolgáló szöveges nézet.

# 3. onCreateViewHolder() – Új listaelemek létrehozása

```
kotlin
MásolásSzerkesztés
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
UserViewHolder {
    val view =
LayoutInflater.from(parent.context).inflate(R.layout.list_item, parent,
false)
    return UserViewHolder(view)
}
```

### Mi történik itt?

1. **`LayoutInflater.from(parent.context)`** – Az `LayoutInflater` felelős a `list_item.xml` elrendezés átalakításáért egy megjeleníthető `View` objektummá.
2. **`inflate(R.layout.list_item, parent, false)`**
   - A `R.layout.list_item` az egyedi listaelem elrendezése, amely a listában jelenik meg.
   - A **`parent`** az az objektum, amely tartalmazza a RecyclerView elemeit.
   - A **`false`** paraméter azt jelenti, hogy a nézetet nem kell azonnal a szülőhöz csatolni.
3. **Visszatérés:** Létrehozunk egy **UserViewHolder** objektumot, amely tárolja az újonnan létrehozott nézetet.

# 4. onBindViewHolder() – Adatok megjelenítése a nézetben

```
kotlin
MásolásSzerkesztés
override fun onBindViewHolder(holder: UserViewHolder, position: Int) {
```

```
    val user = userList[position]
    holder.nameText.text = user.name
    holder.emailText.text = user.email
    holder.profileImage.setImageResource(user.profileImage)
}
```

### Mi történik itt?

1. **`val user = userList[position]`**
    - Lekérjük az aktuális pozíción lévő felhasználói objektumot a listából.
2. **Nézetek frissítése:**
    - A **`holder.nameText.text = user.name`** – A nézetben megjelenik a felhasználó neve.
    - A **`holder.emailText.text = user.email`** – Megjeleníti az e-mail címet.
    - A **`holder.profileImage.setImageResource(user.profileImage)`** – Beállítja a profilképet.

### Miért fontos ez a lépés?

- Az adatok megfelelő indexhez tartozó nézethez való rendelése.
- A lista gyors frissítése és a hatékony memóriahasználat érdekében a nézetek újrahasznosítása.

# 5. getItemCount() – Az adatlista méretének visszaadása

```
kotlin
MásolásSzerkesztés
override fun getItemCount() = userList.size
```

### Mit csinál ez a függvény?

- Visszaadja a lista méretét, vagyis hogy hány elemet kell megjeleníteni.
- Ezt a RecyclerView automatikusan használja annak meghatározására, hogy hány elemre van szüksége.

## Hogyan működik az egész folyamat?

1. **A RecyclerView meghívja az `onCreateViewHolder` függvényt**, amikor új elemre van szükség a képernyőn.
2. **Az `onBindViewHolder` beállítja az adatokat**, amikor egy elem újra felhasználásra kerül vagy első alkalommal megjelenik.
3. **A ViewHolder tárolja a nézetek referenciáit**, hogy ne kelljen minden egyes alkalommal újra keresni őket a DOM-ban.
4. **Az újrahasznosítás miatt kevesebb memóriát fogyaszt**, és jobb teljesítményt biztosít nagy adatlisták esetén.

## Összegzés

- **ViewHolder**: Gyorsítótárazza az egyes listaelemek nézeteit.
- **Adapter**: Az adatok és a nézetek közötti kapcsolatot biztosítja.
- **RecyclerView működése**:
    - Nézetek létrehozása (onCreateViewHolder).
    - Adatok beállítása (onBindViewHolder).
    - Elem méretének visszaadása (getItemCount).

- **Hatékonyság**: A RecyclerView az újrafelhasználás révén jobb teljesítményt biztosít, mint a ListView.

### 5. RecyclerView konfigurálása az Activity-ben (MainActivity.kt)

A fő tevékenység (activity) inicializálja és kezeli a RecyclerView-t.

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val recyclerView: RecyclerView = findViewById(R.id.recyclerView)

        val users = listOf(
            User("Kovács Péter", "peter.kovacs@email.com",
R.drawable.ic_user),
            User("Nagy Anna", "anna.nagy@email.com", R.drawable.ic_user)
        )

        recyclerView.layoutManager = LinearLayoutManager(this)
        recyclerView.adapter = UserAdapter(users)
    }
}
```

**Mi történik itt?**

1. **Adatok létrehozása (Statikus lista)**
2. **LayoutManager beállítása**
   - A `LinearLayoutManager(this)` függőleges görgetést biztosít.
3. **Adapter beállítása a RecyclerView-hoz**

### Összegzés – Hogyan működik a RecyclerView?

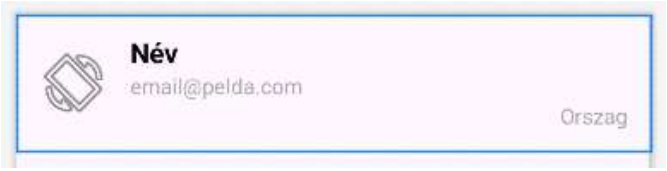
1. **RecyclerView** létrejön az XML-ben.
2. **Az Adapter** kezeli az adatok megjelenítését az egyes elemekben.
3. **A ViewHolder** gyorsítótárazza az egyes listaelemeket.
4. **Az adatok dinamikusan töltődnek be, és az újrafelhasználás javítja a teljesítményt.**

# Gyakorlat (RandomUserList)

A RandomUser lista létrehozásának lépései vázlatosan:

## A lista és listaelem xml fájlok létrehozása

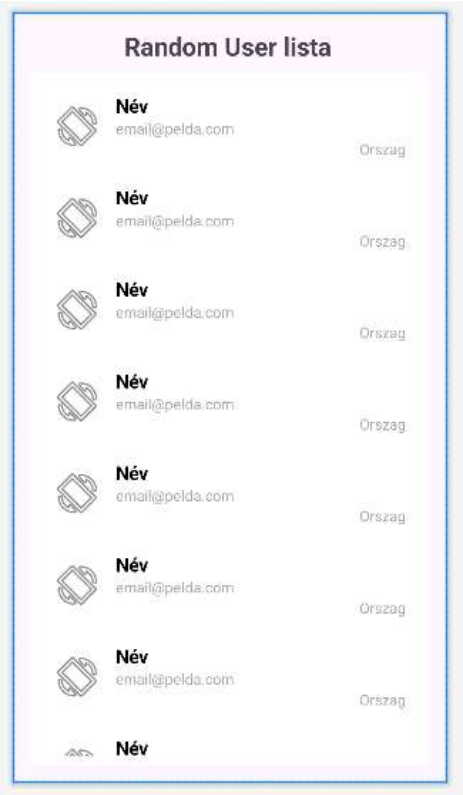
A listaelem:




```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:padding="12dp"
    android:background="?attr/selectableItemBackground"
    android:gravity="center_vertical">

    <ImageView
        android:id="@+id/imageview_profile"
        android:layout_width="50dp"
        android:layout_height="50dp"
        android:src="@android:drawable/ic_menu_always_landscape_portrait"
        android:scaleType="centerCrop"
        android:layout_marginEnd="12dp"
        android:contentDescription="Profilkép"/>

    <LinearLayout
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:orientation="vertical">

        <TextView
            android:id="@+id/textview_name"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Név"
            android:textSize="18sp"
            android:textStyle="bold"
            android:textColor="@android:color/black"/>

        <TextView
            android:id="@+id/textview_email"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="email@pelda.com"
            android:textSize="14sp"
            android:textColor="@android:color/darker_gray"/>

        <TextView
            android:id="@+id/textview_country"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Orszag"
            android:textSize="14sp"
            android:layout_gravity="end"
            android:textColor="@android:color/darker_gray"/>
    </LinearLayout>
</LinearLayout>
```

És a lista a RecyclerView használatával:




```
<LinearLayout
    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:padding="16dp"
    tools:context=".RandomUserListActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Random User lista"
        android:textSize="24sp"
        android:textStyle="bold"
        android:paddingBottom="8dp"
        android:layout_gravity="center"/>

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/randomUserListRecyclerView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="8dp"
        android:background="@android:color/white"
        tools:listitem="@layout/list_item_randomuser"/>
</LinearLayout>
```

## Az API endpoint interfész meghatározása

A Random User API elérhető az alábbi címen. Célszerű paraméterezhető formában használni, hogy a BaseUrl több híváshoz is használható legyen.

https://randomuser.me/api/?results=10

Ellenőrizzük le Postman-ben!!!

A hívás ilyen lesz:

```
interface RandomUserService {
    @GET("/api/")
    suspend fun getRandomUsers(
        @Query("results") results: Int
    ): RandomUserResponse
}
```

*Magyarázat:*

**1. Interfész neve: `RandomUserService`**

- Az interfész egy absztrakció, amely a hálózati kommunikációt írja le.
- Az interfészben definiált függvényeket a Retrofit használja fel a megfelelő HTTP-kérések végrehajtására.

**2. Annotáció: `@GET("/api/")`**

- Ez az annotáció jelzi, hogy az adott függvény egy **HTTP GET** kérést hajt végre.
- Az idézőjelekben szereplő útvonal (`"/api/"`) az API végpontját jelöli, amelyet a Retrofit a base URL-hez fog csatolni.
- Példa base URL lehet: `"https://randomuser.me"`

**3. A függvény: `getRandomUsers`**

```
suspend fun getRandomUsers(
     @Query("results") results: Int
): RandomUserResponse
```

- **`suspend` kulcsszó**:
  - A függvény felfüggeszthető függvényként van megjelölve, ami azt jelenti, hogy koroutinokban hívható meg.
  - Lehetővé teszi az aszinkron hívásokat, anélkül hogy blokkolná a fő szálat.
- **Függvény paramétere:**
  - `@Query("results") results: Int`
    - Az `@Query` annotáció hozzáad egy lekérdezési paramétert az URL-hez.
    - Ha például az `results` értéke 10, akkor a végső kérés így néz ki:

      `https://randomuser.me/api/?results=10`

    - Az API így 10 véletlenszerű felhasználót ad vissza.
- **Visszatérési típus: `RandomUserResponse`**
  - Ez egy adatmodell osztály, amely a szerver válaszát reprezentálja.
  - Az API által visszaadott JSON adatokat a Retrofit ebbe az objektumba alakítja át automatikusan.

## Adatosztályok létrehozása

Az API JSON válaszához létre kell hoznunk az adatok struktúráját, ami megfelel a küldött JSON szekezetének:

```
data class RandomUserResponse(
    val results: List<RUser>
```

```
)

data class RUser(
    val name: Name,
    val location: Location,
    val email: String,
    val picture: Picture
)

data class Name(
    val first: String,
    val last: String
)

data class Location(
    val country: String
)

data class Picture(
    val medium: String
)
```

## Hálózati kérés, adatok letöltése adatosztályba

```
private suspend fun fetchRandomUserList(): List<RUser>? {
    val retrofit = Retrofit.Builder()
        .baseUrl("https://randomuser.me/api/")
        .addConverterFactory(GsonConverterFactory.create())
        .build()

    val randomUserService = retrofit.create(RandomUserService::class.java)

    return withContext(Dispatchers.IO) {
        try {
            val response = randomUserService.getRandomUsers(10)
            response.results
        } catch (e: Exception) {
            Log.e("UserListActivity", "Error fetching user list", e)
            null
        }
    }
}
```

### 1. Függvény neve és típusa

```
private suspend fun fetchRandomUserList(): List<RUser>?
```

- **`private`**: A függvény csak az adott osztályon belül érhető el, tehát más osztályból nem lehet közvetlenül meghívni.
- **`suspend`**: Az aszinkron működés támogatása érdekében a függvény felfüggeszthető (koroutinnal kell hívni).
- **Visszatérési érték: `List<RUser>?`**
  - Egy felhasználók listáját adja vissza (`List<RUser>`), vagy ha hiba történik, akkor `null` értéket.

### 2. Retrofit objektum létrehozása

```
val retrofit = Retrofit.Builder()
    .baseUrl("https://randomuser.me/api/")
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

**Mit csinál ez a rész?**

- **`Retrofit.Builder()`**:
  - Létrehoz egy Retrofit példányt, amelyet hálózati kérések végrehajtására használunk.
- **`baseUrl("https://randomuser.me/api/")`**:
  - Beállítja az API alap URL-jét, amelyhez a függvényben definiált végpontokat fűzi hozzá.
  - Fontos, hogy a base URL mindig "/" jellel végződjön.
- **`addConverterFactory(GsonConverterFactory.create())`**:
  - A JSON válaszokat automatikusan Kotlin adatobjektumokká alakítja a `GsonConverterFactory` segítségével.
- **`build()`**:
  - A Retrofit példány végleges létrehozása.

### 3. Szolgáltatás interfész példányosítása

```
val randomUserService = retrofit.create(RandomUserService::class.java)
```

- Itt a Retrofit segítségével létrehozzuk a `RandomUserService` interfész implementációját.
- Az API meghívásakor a `getRandomUsers()` függvény hívható lesz az API végpontokkal.

### 4. A hálózati kérés végrehajtása a háttérszálon

```
return withContext(Dispatchers.IO) {
```

- **`withContext(Dispatchers.IO)`**:
  - A kód ezen a ponton átvált az IO (Input/Output) szálra, amely a hosszú futásidejű műveletekhez (például hálózati hívásokhoz) van optimalizálva.
  - Ezzel biztosítjuk, hogy a fő szál ne blokkolódjon a hálózati kérés során.

### 5. API hívás és hiba kezelés

```
try {
    val response = randomUserService.getRandomUsers(10)
    response.results
} catch (e: Exception) {
    Log.e("UserListActivity", "Error fetching user list", e)
    null
}
```

**`try` blokk:**

- Az API-t hívjuk meg az interfész függvényével:

  ```
  val response = randomUserService.getRandomUsers(10)
  ```

  - Az API lekérdez 10 véletlenszerű felhasználót.

    - A szerver válasza egy `RandomUserResponse` objektum lesz, amely tartalmazza a `results` listát.
- Visszaadjuk az API válaszából a felhasználók listáját:

```
response.results
```

**`catch` blokk:**

- Ha bármilyen hiba történik a hálózati kérés során (például nincs internet, időtúllépés), az `Exception` elkapásra kerül.
- A hiba naplózása az Android Log rendszerével:

```
Log.e("UserListActivity", "Error fetching user list", e)
```

- Sikertelenség esetén a függvény `null` értéket ad vissza.

## RecyclerView adapter létrehozása

### 1. Az osztály definiálása

```
class RandomUserAdapter(private val userList: List<RUser>) :
RecyclerView.Adapter<RandomUserAdapter.RandomUserViewHolder>()
```

- **`RandomUserAdapter`**: Ez az adapter osztály felelős a `RecyclerView` listaelemeinek kezeléséért.
- **`private val userList: List<RUser>`**:
    - Az adapter egy **felhasználók listáját** (`RUser`) kapja bemenetként, amelyet megjelenít.
    - `RUser` az API válaszából származó adatmodell.
- **`RecyclerView.Adapter<RandomUserAdapter.RandomUserViewHolder>()`**:
    - Az adapter a `RecyclerView.Adapter` osztályból származik, amely generikus típusparaméterként megkapja a saját `ViewHolder`-jét.

---

### 2. ViewHolder osztály

```
class RandomUserViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
    val profileImage: ImageView =
itemView.findViewById(R.id.imageview_profile)
    val nameText: TextView = itemView.findViewById(R.id.textview_name)
    val emailText: TextView = itemView.findViewById(R.id.textview_email)
    val countryText: TextView =
itemView.findViewById(R.id.textview_country)
}
```

- **Mi az a `ViewHolder`?**
    - A `ViewHolder` minta segít a `RecyclerView`-ban hatékonyan kezelni az elemeket.
    - Minden egyes listaelem (sor) nézetét tartalmazza, és elmenti a hivatkozásokat az egyes UI elemekre, így nem kell minden alkalommal újra keresni őket.
- **Tagváltozók:**

- **`profileImage`**: Az adott felhasználó profilképét megjelenítő `ImageView`.
- **`nameText`**: A felhasználó neve.
- **`emailText`**: A felhasználó e-mail címe.
- **`countryText`**: A felhasználó országa.

---

## 3. View létrehozása (`onCreateViewHolder`)

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
RandomUserViewHolder {
    val view =
LayoutInflater.from(parent.context).inflate(R.layout.list_item_randomuser,
parent, false)
    return RandomUserViewHolder(view)
}
```

- **Mi történik itt?**
  - Ez a függvény akkor hívódik meg, amikor a `RecyclerView`-nak új nézetet kell létrehoznia.
  - **`LayoutInflater.from(parent.context)`**:
    - A `LayoutInflater` betölti a `list_item_randomuser.xml` layout fájlt, amely egy listaelem (sor) megjelenítéséért felelős.
  - **`inflate()`**:
    - Az `inflate()` metódus a layout XML-t tényleges `View` objektummá alakítja.
  - **`return RandomUserViewHolder(view)`**:
    - Létrehozunk egy új `ViewHolder` példányt, amely tartalmazza az inflált nézetet.

---

## 4. Adatok betöltése a nézetekbe (`onBindViewHolder`)

```
override fun onBindViewHolder(holder: RandomUserViewHolder, position: Int)
{
    val user = userList[position]
    holder.nameText.text = "${user.name.first} ${user.name.last}"
    holder.emailText.text = user.email
    holder.countryText.text = user.location.country

    Glide.with(holder.itemView.context)
        .load(user.picture.medium)
        .placeholder(R.drawable.user)
        .error(R.drawable.ic_launcher_background)
        .into(holder.profileImage)
}
```

- **Mi történik itt?**
  - Ez a függvény tölti fel az egyes nézeteket az adott felhasználó adataival.
  - **`val user = userList[position]`**:
    - A `userList`-ből kiválasztjuk a `position` indexű felhasználót.
- **Adatok megjelenítése:**
  - **Név:**

    ```
    holder.nameText.text = "${user.name.first} ${user.name.last}"
    ```

A felhasználó keresztneve és vezetékneve jelenik meg.

- **E-mail:**

```
holder.emailText.text = user.email
```

- **Ország:**

```
holder.countryText.text = user.location.country
```

- **Képek betöltése Glide segítségével:**

```
Glide.with(holder.itemView.context)
    .load(user.picture.medium)
    .placeholder(R.drawable.user)
    .error(R.drawable.ic_launcher_background)
    .into(holder.profileImage)
```

  - **`Glide`**: Egy képek betöltésére szolgáló könyvtár.
  - **`load(user.picture.medium)`**: Betölti a felhasználó közepes méretű profilképét az adott URL-ről.
  - **`placeholder(R.drawable.user)`**: Egy alapértelmezett kép jelenik meg, amíg a tényleges kép betöltődik.
  - **`error(R.drawable.ic_launcher_background)`**: Ez a kép jelenik meg, ha a kép betöltése sikertelen.

---

### 5. Lista méretének visszaadása (`getItemCount`)

```
override fun getItemCount() = userList.size
```

- Ez a függvény visszaadja, hogy hány elemet tartalmaz a `RecyclerView`.
- **`userList.size`**: A felhasználók listájának hossza.

## Felhasználók listájának megjelenítése