# עיצוב האפליקציה

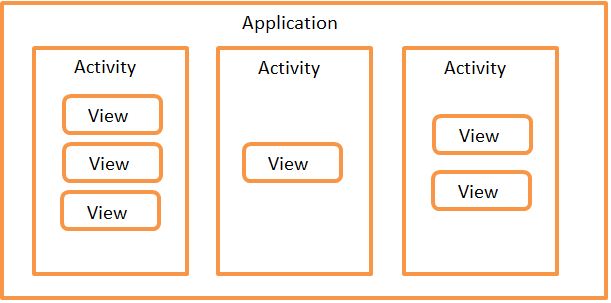
## איך מתחילים

יוצרים פרויקט חדש ב-Android Studio. יש לתת לפרויקט שם ולבחור מספר גרסה. בבחירת הגרסה של אנדרואיד שבה נפתח את המוצר, מצד אחד ככל שהגרסה יותר מתקדמת ככה נוכל להשתמש ביותר כלים מתקדמים בקוד שלנו, אך מצד שני כשנפיץ את האפליקציה רק מי שיש לו גרסה זו ומעלה יוכל להשתמש בה. לכן מומלץ בדר"כ להיות איפשהו באמצע. בעת כתיבת שורות אלו הגרסה המומלצת היא 5.1. נוכל בנוסף גם לבחור תבנית בסיסית שעליה אנו רוצים לעבוד.

## קבצי הפרויקט

* קבצי manifests - קבצים המתארים תכונות של האפליקציה והגדרה של הרכיבים השונים המוכלים בה. ניתן ליצור אוטומטית.
* קבצי Java - כל קבצי ה-java של הפרויקט. זה מה שרץ כשנכנסים לאפליקציה.
* res/drawable - כל קבצי התמונות.
* res/layout - קבצי xml המגדירים ממשק משתמש.
* res/mipmap -
* res/values - קבצי xml שונים עבור הגדרות של צבעים, מחרוזות ועוד נתונים שונים.
* קבצי Gradle - קבצים המגדירים את כל מה שקשור לקימפול האפליקציה. נוצר אוטומטית

## Activity

כל אפליקציה מורכבת מכמה מסכים שכל אחד מהם נקרא activity. כל אקטיביטי מורכב ממספר רכיבים הנקראים views. view יכול להיות כפתור, טקסט פלט, קלט של טקסט, וכו'. ניתן להוסיף אקטיביטי על ידי: File->new->Activity.

כל אקטיביטי מורכב משתי קבצים: קובץ java שאחראי על הלוגיקה שלו והפעולות שקורות במסך, וקובץ xml של ממשק המשתמש האחראי על מבנה המסך - Layout.

## Views

אבן הבניין הבסיסית עבור ממשק המשתמש היא אובייקט View, נקרא לה גם "רכיב", שתופס שטח מלבני על המסך ואחראי על ציור וטיפול באירועים. view משמש ליצירת רכיבי ממשק משתמש אינטראקטיביים.

### סוגי view

TextView - טקסט על המסך שלא ניתן לשנותו.

EditText - חלון שבו המשתמש יכול להכניס קלט.

Buttons - כפתור ללחיצה.

ImageView - תמונה על המסך.

### מאפייני view

לכל view על המסך ניתן להגדיר מאפיינים רבים בקובץ ה-xml של האקטיביטי בתוכו נמצא. ישנם מאפיינים הייחודיים לסוג מסוים של views ואינם רלוונטיים לסוגים אחרים.

* **id** - לכל רכיב יש id ייחודי.
* **layout\_width** - רוחב הרכיב ביחידות dp. ניתן גם לתת ערכים כמו wrap\_content שאומר לעטוף את תוכן הרכיב במדויק, או match\_parent שאומר להשתלט על כל המקום לצדדים שניתן ל- ViewGroupשבתוכו הרכיב הנוכחי.
* l**ayout\_height** - גובה הרכיב ביחידות dp. אפשר גם לתת אותן ערכים כמו ברוחב.
* **textSize** - גודל הטקסט בתוך הרכיב. ניתן ביחידות של sp.
* **background** - צבע או תמונה שיהיו ברקע.
* **textColor** - צבע הטקסט.
* **gravity\_layout**- מרכוז הרכיב לימין, למרכז או לשמאל.
* **gravity** - מרכוז הטקסט או התמונה שבתוך הרכיב למרכז.
* **margin**  - גודל השוליים מה-view שלפני או אחרי.
* **padding** - שוליים של התוכן (לדוגמא הטקסט) מגבולות הרכיב.
* **src** - ברכיב המציג תמונה שדה זה מכיל את מיקום התמונה.

**SP** (scale-independent pixels) - מתאים את הגודל לפי הרזולוציה של המסך.

**DP** (density-independent pixels) - גם מתאים לפי הרזולוציה של המסך, אבל גם מתחשב בהעדפות הגודל של המשתמש.

דוגמה: להגדרת מספר רכיבים בקובץ xml. כל שורה מתחילה במילה android כדי לציין את הסכמה ממנה מגיע המידע.

<TextView  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="ברוכים הבאים לבריאותא"

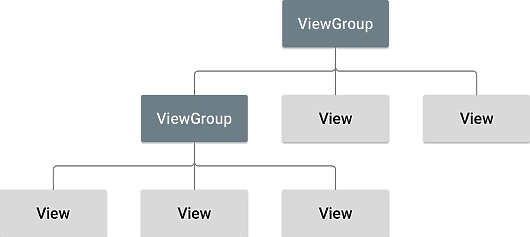
android:textSize="40dp"  
 android:textColor="#58d41e"  
 android:layout\_gravity="center"  
 android:gravity="center"  
 android:id="@+id/txtwelcome" />

<Button  
 android:layout\_width="250dp"  
 android:layout\_height="wrap\_content"  
 android:text="כניסה ללקוחות"

android:textSize="15sp"  
 android:layout\_gravity="center"  
 android:id="@+id/btnlogin" />

<ImageView  
 android:id="@+id/imageView"  
 android:layout\_width="300dp"  
 android:layout\_height="300dp"  
 android:src="@drawable/briuta"  
 android:layout\_gravity="center"/>

## Layout

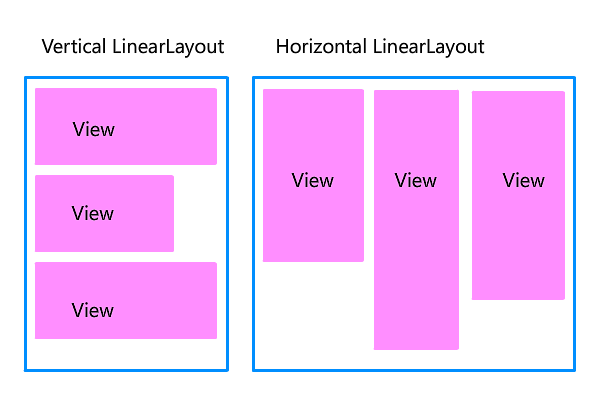
ניתן לאגד מספר רכיבים עם מאפיינים זהים תחת ViewGroup אחד. ViewGroup הוא מיכל בלתי נראה המחזיק קבוצה של רכיבים ואף יכול להכיל בתוכו גם ViewGroup.

Layout מגדיר את מאפייני הפריסה של ViewGroup על המסך. כל האלמנטים ב-layout בנויים מהיררכיה של view ו-viewGroup. יצירה והגדרה של layout יכולה להיות דרך ה-GUI וגם דרך הקוד Java.

### סוגים של Layouts

* LinearLayout - כל הרכיבים במסך מסודרים אחד אחרי השני באותו כיוון לפי סדר הגדרתם. ניתן לבחור האם Layout זה יהיה אנכי (vertical) או אופקי (horizontal). נבחר איזה מהם באמצעות מאפיין orientation.

<LinearLayout xmlns:android=”http://schemas.android.com/apk/res/android”

 android:layout\_width="fill\_parent"  
 android:layout\_height="fill\_parent"  
 android:orientation=" horizontal “

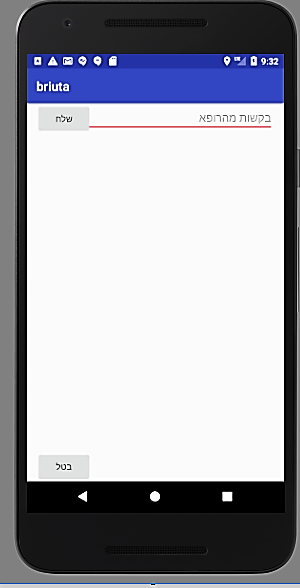
...

</LinearLayout>

* RelativeLayout - פרישה יחסית של כל הרכיבים. יש להגדיר ייחוס לכל רכיב ביחס לרכיבים אחרים. רכיב ימצא מתחת לרכיב הורה שלו, מעל לרכיב הבן שלו, ומשמאל/ימין לרכיב אח שלו.

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

    android:layout\_width="fill\_parent"

    android:layout\_height="fill\_parent"

    android:paddingLeft="16dp"

    android:paddingRight="16dp”>

  <EditText

        android:id="@+id/name"

        android:layout\_width="fill\_parent"

        android:layout\_height="wrap\_content"

        android:hint="@string/reminder”/>

<Button

            android:layout\_width="wrap\_content"

            android:layout\_height="wrap\_content"

            android:text="שלח"

            android:id="@+id/button"

            android:layout\_alignLeft="@+id/name”/>

  <Button

            android:layout\_width="wrap\_content"

            android:layout\_height="wrap\_content"

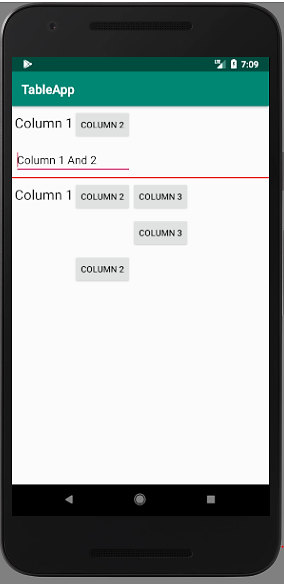
            android:text="בטל"

            android:id="@+id/button2"

            android:layout\_alignParentBottom="true”/>

</RelativeLayout>

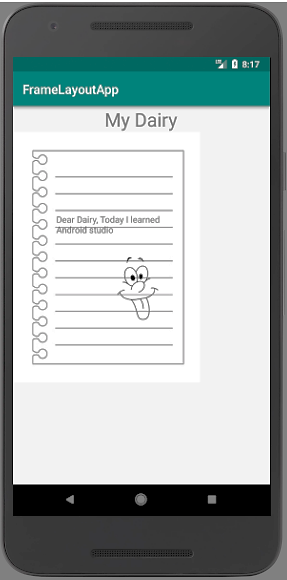
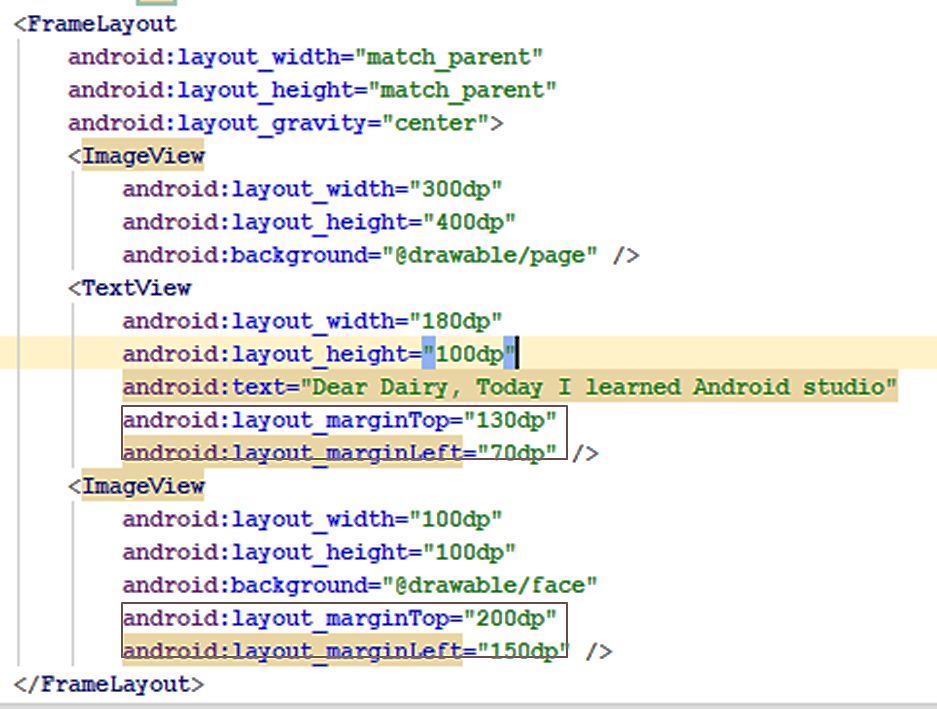
* TableLayout - כל הרכיבים מסודרים בתוך טבלה המחלקת את המסך לשורות. כל תגית <TableRow> היא שורה בטבלה. עבור כל שורה כזו צריך להגדיר רוחב וגובה. כל שורה יכולה להכיל 0 או יותר תאים. בכל תא יהיה רכיב בודד.



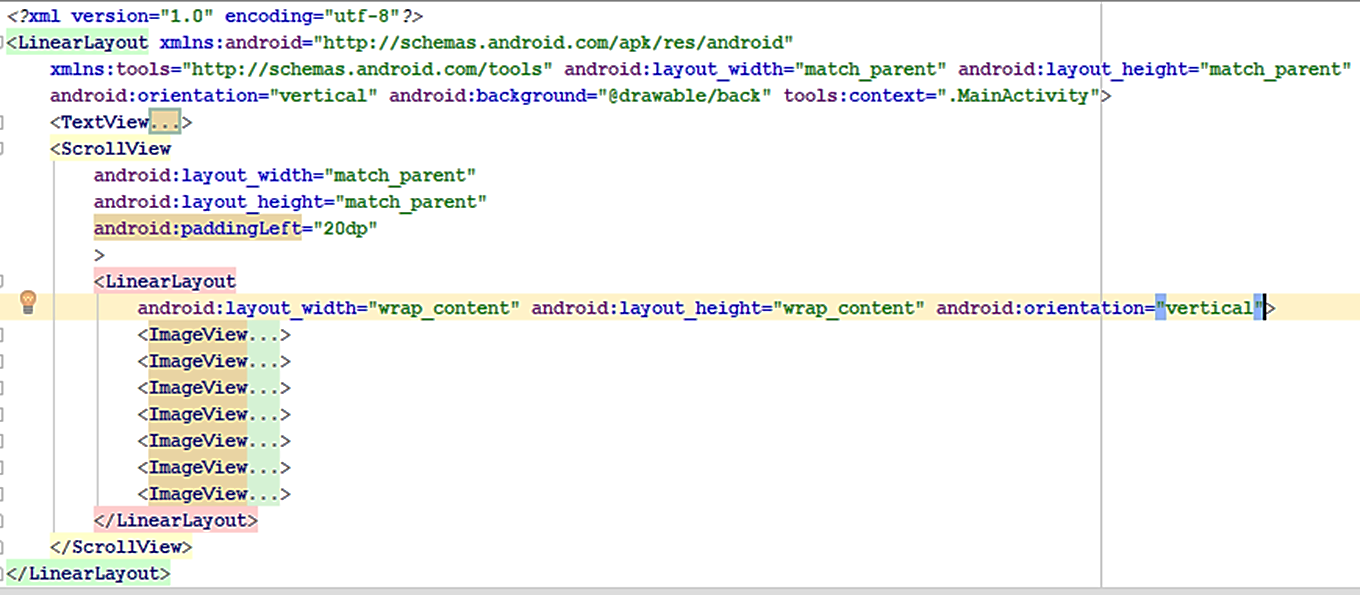
* ConstraintLayout - כל רכיב מקבל שם ומוגדר באופן יחסי לרכיבים אחרים במסך (בדומה ל-RelativeLayout). אך בנוסף ניתן לקבוע מה השוליים בינו לבין כל רכיב אחר שהוגדר יחסית אליו. צריך שכל רכיב יהיה מיוחס לפחות לשני רכיבים, אחד מאוזן והשני מאונך. ב-layout הזה מומלץ לעבוד עם עורך הפריסה ולא ישירות עם ה-xml. שיטה זו מיועדת למסכים מורכבים. זמן העלייה של מסך מורכב שמעוצב בשיטה זו יהיה קצר באופן יחסי לשיטות האחרות.



* FrameLayout - שיטה שבה מציבים רכיבים אחד על השני. לכל רכיב צריך לציין מה המרחק מהשוליים מכל 4 הצדדים.



## ScrollView

ניתן להגדיר view בתור ScrollView. פעולה זו מאפשרת למשתמש לגלול בתוך הרכיב. ניתן להגדיר גלילה אנכית או אופקית. בתוך ה-ScrollView חייבים להגדיר LinearLayout שיכול להיות אופקי או אנכי ובתוכו נמקם את כל הרכיבים שאנו וצים לגלול עליהם.

## למה לשים מחרוזות ב-res/values

אנו מגדירים צבעים, תמונות, עיצוב וכו' ב-res/values. נהוג שגם מחרוזות שבהם אנו משתמשים באפליקציה יהיו שמורים שם בקובץ string.xml ולא כתובים בקוד. ניתן להתעלם מכך אך יש לכך מספר סיבות:

1. שנוכל לגשת אליהם בקלות בלי שנצטרך כל פעם לרשום אותם מחדש.
2. במקרה שנרצה להתאים את האפליקציה לשפה אחרת נוכל ליצור תיקיית משאבים מרובת שפות, כך שהשפה שתשתנה באמצעות Locale תישמר באופן דינמי על ידי מערכת ההפעלה בזמן הריצה בהתאם לתיקיית משאבי השפה.
3. אם בקוד יהיה כתוב את כל המשפטים וההגדרות האחרות, אזי בטעינת האפליקציה ייטען גם מידע זה. ואילו כאשר זה בתוך קובץ ניגשים אליו בזמן ריצה.

4) מחרוזות לא מסתבכות בקוד ומשאירות אותו ברור וקל לתחזוקה.

# מימוש

## תכנות מונחה אירועים

פיתוח אפליקציה נעשית לפי עקרון של "תכנות מונחה אירועים", שבו כותבים בתוכנה פונקציות שיופעלו בתגובה לפעולות שביצע המשתמש, הנקראות אירועים (Events), כמו לחיצה על העכבר או הקשה על המקלדת. כדי שאכן המערכת תזהה שקרה אירוע כלשהו על רכיב מסוים, צריך להפעיל על הרכיב listener מתאים כדי שיאזין להתרחשות מאורע זו.

## Activity

במימוש האפליקציה כל אקטיביטי הוא מחלקה היורשת מ-AppCompatActivity. לכל אפליקציה יש מחלקה אחת של MainActivity שהיא התוכנית הראשית וממנה מתחילה ריצת האפליקציה.

### שלבי העבודה במימוש Activity

1. הגדרת הרכיבים בתור שדות של המחלקה.
2. קישור בין הרכיבים שיש ל-XML לאובייקטים שיצרנו.
3. הגדרת listeners מתאימים על כל רכיב.
4. מימוש הפונקציות שיגיבו לאירועים שהגדרנו.

public class MainActivity extends AppCompatActivity **implements View.OnClickListener** {

Button login;

Button register;

TextView welcome;

@Override  
 **protected void onCreate**(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_main*);

login=(Button)findViewById(R.id.*btnlogin*);  
 register=(Button)findViewById(R.id.*btnregister*);  
 welcome=(TextView)findViewById(R.id.*txtwelcome*);  
 **login.setOnClickListener(this);  
 register.setOnClickListener(this);**  
 }  
 @Override  
 **public void onClick(View view)** {

if (view == login)

welcome.setText("שלום לך לקוח רשום");

if (view == register)

welcome.setText("כיף להצטרף לבריאותא");

}  
}

### Back Stack

בכל אפליקציה יש בדרך כלל מספר Activities. כל ה-Activities נשמרים במחסנית, הנקראת backStack, כך שהאקטיביטי הפעיל תמיד נמצא בראש המחסנית. כל אקטיביטי שנפתח על גבי אחר, האקטיביטי הקודם נכנס ל-backStack, כך שנוכל לחזור למסך הקודם בלחיצה על back בנייד שלנו. זה קורה אוטומטית ואנחנו לא צריכים להגדיר את זה. ישנם מקרים שכן נרצה לגעת במידע שיש ב-backStack, נראה זאת בהמשך כשנראה איך פותחים דפים נוספים ועוברים אליהם באפליקציה.

## Events

דוגמאות לאירועים:

* onCreate() – נקרא כאשר האקטיביטי נוצר. חייבים לממש בכל אקטיביטי.
* onClick() – לחיצה רגילה על העכבר.
* onLongClick() – לחיצה ארוכה.
* onFocusChange() -
* onKey() – לחיצה על המקלדת.
* onTouch()
* onMenuItemClick() – לחיצה על אפשרות בתפריט.
* onCreateContextMenu()

### מימוש פונקציה ל-event

פונקציה שמוגדרת לEVENT צריכה להיות public, לקבל אובייקט View, ולהחזיר void. ישנם שלוש שיטות כיצד לממש שיטה כזו:

1. פונקציה חדשה כמו שראינו בדוגמה בסעיף קודם על האירוע onClick. זוהי השיטה הנוחה והנפוצה ביותר.
2. פונקציה אנונימית המוגדרת בתוך ה-listener שהפעלנו על view כלשהו. צריך להגדיר כל Event על כל רכיב בנפרד. שיטה זו שימושית כאשר משתמשים ב-Fragment או כאשר רוצים לאתחל את הרכיב תוך כדי ריצה.

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);  
 setContentView(R.layout.activity\_main);  
 Button button1 = (Button) findViewById(R.id.entranceBtn);

button1.setOnClickListener(new View.OnClickListener() {

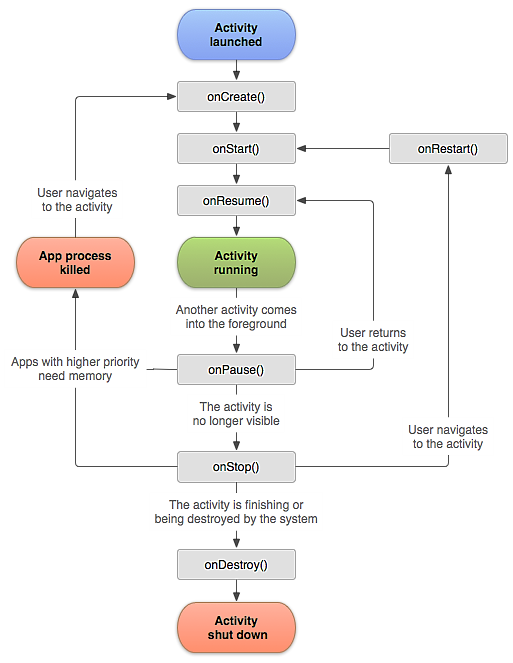
public void onClick(View v) {

// Do something in response to button click

}  
 });  
}

1. להגדיר בתוך קובץ ה-XML של האקטיביטי מי הפונקציה שתופעל כאשר יתרחש event כלשהו, כאשר הפונקציה שתופעל מוגדרת בתוך הקובץ Java. שיטה זו לא מומלצת שכן הקוד לא קריא במצב כזה ולא ברור מתי תיקרא הפונקציה.

## מחזור חיים של Activity

הסבר מפורט על כל הפונקציות של מחזור החיים [כאן](https://developer.android.com/guide/components/activities/activity-lifecycle).

## Intent

זהו אובייקט המאפשר לקשר בין Activities שונים באותה אפליקציה ואף יכול לקשר בין אפליקציות שונות באותו התקן (כמו מצלמה, אנשי קשר, דפדפן וכו'). מאפשר בנוסף להעביר מידע בין רכיבים אלו. יש שני סוגים של intent. הראשון הוא explicit intent, שבו מצינים במפורש לאן רוצים לעבור, והשני הוא implicit intent, שבו מציינים איזה שירות רוצים אך לא מצוין במפורש איזה רכיב לבחור המספק שירות זה. המידע שניתן להכניס בתוך intent הוא:

* **Component Name** - לאן אנחנו רוצים לעבור. לא חובה, אבל קריטי ב-intent מפורש.
* **Action Name** - מחרוזת המציינת את הפעולה הגנרית שתתבצע. אפשר לכתוב את ה-action לבד, או להשתמש בפעולות קיימות. דוגמאות לכמה נפוצות: ACTION\_VIEW, ACTION\_SEND. נדבר עליהם בהמשך.
* **Data -** נתונים שאנחנו רוצים להעביר. בדר"כ זה קשור ל-action שאותו בחרנו לעשות.
* **Category** - מידע נוסף על סוג הרכיב שאנחנו רוצים לפתוח, זה לא כ"כ שימושי, בדר"כ מסתדרים מצויין בלעדיו.
* **Extras** - מידע נוסף שקיים בדף הנוכחי ואנחנו רוצים להעביר למסך הבא שיפתח.
* **Flags** - מידע למערכת האנדרואיד איך לפתוח את האקטיביטי הבא, ואיך לטפל בה לאחר שנפתחה (לדוגמא האם להוסיף אותה לרשימת הפעולות האחרונות).

## Explicit Intent

מציין איזה אקטיביטי או אפליקציה צריך לפתוח באופן מפורש. השימוש הנפוץ ביותר הוא כדי לעבור לאקטיביטי הבא:

public void onClick(View v){

if(btnSave == v){

Intent intent=new Intent(this, SecondActivity.class);

startActivity(intent);

}

}

ניתן להוסיף מידע לאקטיביטי הבא שאנו עוברים אליו על ידי הוספה לאובייקט ה-intent מידע לפני שמפעילים אותו באמצעות startActivity.

intent.putExtra("fname", etFname.getText().toString());

באקטיביטי החדש נוכל לשלוף מידע זה על ידי שיטת getIntent שמחזירה את ה-intent שהביא לפתיחה של אקטיביטי נוכחי.

Intent intent = getIntent();

String fname = intent.getExtras().getString("fname");

### החזרת מידע לאקטיביטי המקורי

אם נרצה להפעיל אקטיביטי שלאחר שיסתיים יחזיר מידע לאקטיביטי שהפעיל אותו, נוכל לעשות זאת באמצעות intent. הפעלת ה-intent במקרה זה נעשית באופן מיוחד על ידי שיטת startActivityForResult. המקבלת intent ו-requestCode. בנוסף, יש לממש פונקציית onActivityResult שתופעל כאשר האקטיביטי שפתחנו עם startActivityForResult יסתיים ונחזור עם המידע לאקטיביטי המקורי. בתוך פונקציה זו יש להבחין בין ה- requestCodeהשונים למקרה שיש מספר Activities שאנחנו מחכים לתוצאות מהם. resultCode מציין האם הבקשה חזרה תקינה או לא, ו-data מחזיר את המידע שרצינו לקבל. דוגמה לאקטיביטי שמפעיל מצלמה ומקבל את התמונה שמצלמת.

public void onClick(View v){

if(btnSave == v){

Intent intent=new Intent(this, SecondActivity.class);

startActivityForResult(intent,0);

}

}

protected void onActivityResult(int requestCode, int resultCode, Intent data){

super.onActivityResult(requestCode, resultCode, data);

if(requestCode==0){

if(resultCode==RESULT\_OK){

bitmap= (Bitmap)data.getExtras().get("data");

iv.setImageBitmap(bitmap);

Toast.makeText(this, "Result ok", Toast.LENGTH\_LONG).show();

}

else

Toast.makeText(this, "Result canceled", Toast.LENGTH\_LONG).show();

}

}

## Implicit Intent

זהו intent שבו רומזים איזה שירות אנו מעוניינים. את הרמזים כותבים ב-intent-filter והם יכולים להיות מסוג: action, category, data.

## Context

זהו אובייקט שמייצג תמונת עולם של מצב כלשהו באפליקציה. כאשר יוצרים אובייקט חדש (בד"כ אקטיביטי חדש) לעיתים נרצה לתת לו את תמונת העולם שהייתה לפני היצירה שלו. לפעמים נרצה תמונת עולם גדולה ולפעמים תמונת עולם קטנה יותר. להעברת ה-context יש שני שימושים עיקריים:

1. לתת מידע על חלק אחר של האפליקציה, בפרט האקטיביטי הקודם שממנו הגענו.
2. נקודת גישה למשאבי מערכת, כמו משאבים גרפים, דאטה בייס, ועוד. וכן גישה לשירותים שאליהם נרשמנו עד עכשיו.

יש שני סוגים של context:

**Activity Context** - מכיל מידע מצומצם שבדרך כלל יעניין רק את הילדים של האקטיביטי הזה. יש לו את אותו מחזור חיים כמו לאקטיביטי. לדוגמה, תבנית (THEM), אוסף צבעים, גופנים וכו', אם יש לנו כאלו שונים לכל אקטיביטי ניתן הקשר של האקטיביטי הזה.

**Application Context** - הוא מכיל מידע שרלוונטי לכל חלקי האפליקציה. יש רק אחד כזה. נשתמש בזה כאשר נרצה להעביר מידע דרך ה-context לאובייקט שהוא לא ילד ישיר של אקטיביטי נוכחי, או כאשר אנחנו רוצים אורך חיים יותר ארוך מה-context הנוכחי. באופן כללי נשתמש בו רק לדברים שנוגעים להרבה חלקים באפליקציה ולא סתם ככה. לדוגמה, אם נרצה לאתחל ספריה שבשימוש ע"י הרבה חלקים באפליקציה ודורשת context, ניתן את Application Context.

### פונקציות של context

* this – מתייחס לאובייקט הנוכחי. הדרך הפשוטה ביותר להעביר את ה-activity context הנוכחי.
* View.getContext() – לקבלת context בתוך activity קיים (זהה לאפשרות הראשונה).
* ContextWrapper.getBaseContext() – כאשר יוצרים אקטיביטי מתוך אקטיביטי קיים, מאפשר לקחת את ה-context של האקטיביטי המקורי דרך הפונקציה הזאת.
* Activity.getApplicationContext() - לקבלת context של כל האפליקציה.

## Toasts

זהו סוג של הודעה פשוטה שאנו יכולים להציג למשתמש. הודעת Toast תופסת רק את הגודל של ההודעה עצמה והיא נעלמת אוטומטית לאחר זמן קצוב. המסך הנוכחי נשאר פתוח ואינטראקטיבי כל משך ההודעה. כדי ליצור Toast נשתמש בשיטה makeText המקבלת context, הודעה, וזמן שאנו רוצים שההודעה תופיע. השיטה מחזירה אובייקט Toast מאותחל שניתן להציגו על ידי שיטת show().

Toast.makeText(this, "This is a toast", Toast.LENGTH\_LONG).show();

## Manifest

קובץ מסוג XML, נקרא גם "קובץ ההצהרות". יש קובץ אחד כזה הנמצא בתיקייה הראשית של הפרויקט. מציג מידע הכרחי על האפליקציה למערכת האנדרואיד שהיא צריכה לפני שהיא מריצה את האפליקציה. קובץ ה-manifest נוצר אוטומטית אך צריך להבין מה כתוב שם כי גם ניתן לשנות אותו. קובץ ה-manifest מכיל:

* אייקון ושם האפליקציה.
* Activities - איזה אקטיביטי יש באפליקציה. ותכונות מיוחדות שיש להם, כמו מי ה-main, כיצד להציג את האקטיביטי (מצב מאוזן לדוגמה).
* מחלקות מיוחדות – כמו לדוגמה מחלקה המשמשת Receiver לאירועים.
* מהם ההרשאות שהאפליקציה צריכה.
* Uses feature - גורם לכך שבחנות של גוגל האפליקציה תוצג רק עבור מכשירים המתאימים להשתמש באפליקציה.
* Uses permission - מבקש הרשאה להשתמש בסנסורים של המכשיר (מצלמה, מיקרופון, מיקום וכו').
* צורכי האפליקציה מבחינת תוכנה וחומרה.

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

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

package="merav.menuapplication">

<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">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

<activity android:name=".SecondActivity"></activity>

</application>

</manifest>

## קובץ build.gradle

באופן כללי Gradle זוהי מערכת קוד פתוח המשמשת לקימפול קוד. מבוססת על Ant ו-Maven. יש שני קבצים כאלו: אחד ברמת הפרויקט והשני ברמת המודל. מאפשר להגדיר סוגים שונים של הרצה באפליקציה. קובץ זה באנדרואיד מכיל:

* פרטים על האנדרואיד - גרסת אנדרואיד, id, sdk, config וכו'.
* מכיל את כל התלויות (dependencies).

R. - מזהה של הקובץ עם משאבים לכל המקורות. לאחר מכן יש לבחור את ה-layout המתאים.

Bundle - משמש להעברת מידע בין Activities.

## Permissions

שיטה לקבל הרשאות גישה של האפליקציה לרכיבים אחרים במכשיר (חומרה או תוכנה). עד גרסה 6 של אנדרואיד ההרשאות ניתנו על ידי המשתמש בעת ההתקנה והן ניתנות לניהול דרך הגדרות האפליקציה. מגרסה 6 והלאה הרשאות קריטיות ניתנות לאישור המשתמש גם בזמן ריצה.

הרשאות מסווגות ל-2 סוגים:

הרשאות רגילות - אינטרנט, בלוטוס’, wifi. בהרשאות רגילות המערכת תיתן אישור אוטומטי.

הרשאות מסוכנות - מצלמה, איחסון חיצוני, אנשי קשר, sms, מיקום. בהרשאות מסוכנות המערכת תבקש אישור מהמשתמש.

# Firebase

## הגדרה

שירות רשת היכול להחליף את כתיבת צד השרת במערכת של mobile ואפליקציות web. השירותים שהא מספק הם:

* בסיס נתונים (לא רלציוני) לאחסון מסמכים (Realtime).
* שירות hosting של נתונים וקבצים.
* מערכת אימות מובנית על ידי אימייל/סיסמא או דרך google/facebook/twitter וכו'.
* ניתוח - מדידת אפליקציות, משתמשים, כמויות ועוד.

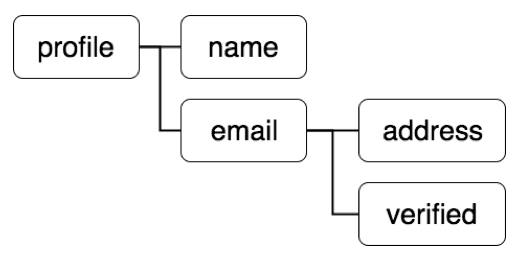
כדי ליצור פרויקט יש להיכנס לאתר [כאן](https://console.firebase.google.com/u/0/?pli=1), ליצור פרויקט ואז לחבר אותו ל-Android Studio לפי ההדרכה באתר. יש לשנות הגדרות של קבצי ה-gradle באפליקציה שלנו.

## מבנה ה-DB

מסד הנתונים ב-Firebase הוא מסד נתונים לא רלציוני שבו כל המידע נשמר בקבצי Json. Json (ראשי תיבות של JavaScript Object Notation) הוא פורמט טקסטואלי קל להבנה, המיועד להעברת מידע בצורה של key : value. ערך ה-value גם כן יכול להכיל key : value, כך שניתן לייצג גם אובייקטים בתוך ה-DB. סוגי המשתנים שניתן לשמור ב-DB הם:

* String
* Long
* Double
* Boolean
* Map<String, Object>
* List<Object>
* Null (empty)
* Java object

כל קובץ Json ניתן לסדר בתור עץ היררכי כמו שניתן לראות בדוגמה הבאה. כך גם המידע מוצג לנו ב-Firebase.

{

"profile" : {

"name" : "Peter",

"email" : {

"address" : "pete@demo.com",

"verified" : true

}

}

}

## קישור ל-DB

בשלב ראשון יש ליצור אובייקט מסוג FirebaseDatabase ולאתחל אותו עם השיטה הסטטית getInstance(). הגישה ל-DB נעשית באמצעות אובייקט DatabaseReference שמצביע לצומת מסוים ב-DB. כדי לאתחל אובייקט זה שיצביע למקום מסוים שנרצה, נפעיל על האובייקט המייצג את ה-DB שלנו את שיטת getReference(), המקבל Path של המיקום שאנו רוצים להצביע עליו. אם לא מכניסים כלום מצביע לשורש ה-DB.

שיטה נוספת שבאמצעותה נוכל להיכנס עמוק יותר ב-DB ביחס לרפרנס, אך לא לשנות אותו, היא שיטת child המקבלת צומת שהיא בן לרפרנס נוכחי ומחזירה רפרנס לצומת הבן.

FirebaseDatabase Database;

DatabaseReference dbRef = database.getReference(); // point to root of DB

dbRef.child("test").child("data").child("message1") // don’t change dbRef

dbRef = database.getReference("/test/data/message1"); // point to specific location

כאמור, כל גישה ל-DB כרוכה בהפניה ל- DatabaseReferenceהכולל בתוכו את כל הצמתים שתחת המיקום שנבחר. יש לשים לב שקינון עמוק מידי יגרום להחזקה של refernces גדולים שעלולים לפגוע ביעילות ב-DB. לכן במקרה של צורך בקינון עמוק יש להחליפו בצומת חדשה ברמה גבוהה יותר.

## שמירת נתונים

לאחר שיצרנו רפרנס למקום מסוים ב-DB נוכל לשמור שם ערכים באמצעות הפעלת שיטת setValue() עליו. ניתן להכניס כל סוג משתנה מאלו שצוינו בסעיף [ב'](#_מבנה_ה-DB). יש לשים לב שכל ערך שהיה שמור באותו רפרנס יידרס לאחר פעולת הכנסה זו. כאשר מכניסים אובייקט, יש לוודא שהמחלקה ממנה נוצר מכילה getters ו-setters לכל שדה פרטי, בנאי ברירת מחדל ובנאי המקבל את כל הפרמטרים של המחלקה. בנוסף, ניתן להכניס אובייקט מסוג CompletionListener שיכיל פונקציה אנונימית שתופעל בסוף השמירה של האובייקט. באמצעות פונקציה אנונימית זו נוכל לדעת האם פעולת השמירה ב-DB עברה בהצלחה, ואיזה פעולה לבצע בכל מצב.

דוגמה לשמירת פרטי לקוח ב-DB והודעה ללקוח האם הפרטים נשמרו או לא.

FirebaseDatabase database = FirebaseDatabase.getInstance();

DatabaseReference dbRootRef = database.getReference();

save.setOnClickListener(this);

....

@Override

public void onClick(View v) {

if (v == save) {

Client client=new Client(name.getText().toString(), birthDate.getText().toString(),"1“,id);

String clientId = dbRootRef.push().getKey();

**dbRootRef.child("clients").child(clientId).setValue(client,completionListener);**

}

}

DatabaseReference.CompletionListener listener = new DatabaseReference.CompletionListener() {

@Override

public void onComplete(DatabaseError dbError, DatabaseReference dbReference) {

if (databaseError != null) // if there is an error inform the user

Toast.makeText(getApplicationContext(),databaseError.getMessage(), Toast.LENGTH\_LONG).show();

else

Toast.makeText(getApplicationContext(),“"נשמר בהצלחה!,Toast.LENGTH\_LONG) .show();

}

};

### שמירה של מספר ערכים במקביל

כאשר רוצים שמספר ערכים יישמרו כולם באותו הזמן (טרנזקציה), כך שלא ייווצר זמן כלשהו שבו חלק יישמרו וחלק לא, נשתמש בפונקציה updateChildren. ניצור אובייקט Map שיכיל את כל המידע שאנו רוצים להכניס ל-DB, כך שהכתובת ביחס לרפרנס היא ה-key ב-Map והערך שנרצה לשמור הוא ה-value. לאחר מכן נכניס את האובייקט Map לרפרנס המתאים באמצעות שיטת updateChildren.

dbRef = database.getReference("/test/data");

Map<String, Object> childUpdates = new HashMap<>();

childUpdates.put("message1/title", "Title 1");

childUpdates.put("message1/content", "Content string 1");

childUpdates.put("message2/title", "Title 2");

childUpdates.put("message2/content", "Content string 2");

**dbRef.updateChildren(childUpdates);**

## מחיקת נתונים

כדי למחוק נתונים מה-DB יש להפעיל על הרפרנס המתאים שיטת removeValue(). דרך מקובלת נוספת היא להשתמש בשיטות שלמדנו בסעיף קודם, אלא שנשמור את הערך null. הפעולה שתתבצע היא דריסת הערך הקיים ושמירת null במקומו. שיטת setValue כדי לשמור null ברפרנס בודד, ושיטת updateChildren שתשמור null בכמה מקומות במקביל.

dbRef.child("message1").child("content").setValue(null);

dbRef.child("message1").child("content").removeValue();

Map<String, Object> childUpdates = new HashMap<>();

childUpdates.put("message1/title", null);

childUpdates.put("message1/content", null);

childUpdates.put("message2/title", null);

childUpdates.put("message2/content", null);

dbRef.updateChildren(childUpdates);

## קריאה וחיפוש

מכיוון ש-firebase הוא Realtime DB, אין דרך לקרוא את הנתונים באופן ישיר אלא צריך להוסיף ל- DatabaseReference listener, שבתוכו יהיו פונקציות אנונימיות שיקראו כשיש שינויים על העץ שהרפרנס הוא השורש שלו, **וכן כאשר מפעילים את הרפרנס בפעם הראשונה**. יש שני סוגים של listeners שאפשר להפעיל:

1. addValueEventListener - יופעל בכל שינוי בחלק העץ שתחת ה-reference שעליו הפעלנו את ה-listener.
2. addListenerForSingleValueEvent - יופעל רק בפעם הראשונה שנעשה שינוי. נשתמש כאשר נרצה לבצע איזו פעולה שגם אם יהיה שינוי לא תשתנה.

בשני סוגי ה-listeners יש לממש את הפונקציה onDataChange שתופעל כאשר יש שינוי ברפרנס. פונקציה זו מקבלת אובייקט dataSnapshot המייצג תמונת מצב של המידע בחלק של העץ שמתחת לרפרנס נוכחי. נשתמש באובייקט זה כדי לקרוא את המידע. כאשר נרצה להפסיק להאזין לשינויים על הרפרנס הנוכחי, או בסיום העבודה, נפעיל עליו את שיטת removeEventListener() שתסיר את כל ה-listeners שהפעלנו.

דוגמה לקוד שקורא הודעה השמורה ב-DB. במידה והקריאה לא הצליחה מדפיס את הסיבה לכך.

ValueEventListener postListener = new ValueEventListener() {

@Override

public void onDataChange(DataSnapshot dataSnapshot) {

// Get Post object and use the values to update the UI

Post post = dataSnapshot.getValue(Post.class);

// ...

}

@Override

public void onCancelled(DatabaseError databaseError) {

// Getting Post failed, log a message

Log.w(TAG, "loadPost:onCancelled", databaseError.toException());

// ...

}

};

mPostReference.addValueEventListener(postListener);

### dataSnapshot

השיטות שניתן להפעיל על אובייקט מסוג dataSnapshot שיעזרו לנו לקרוא את המידע הם:

* getRef - אובייקט המייצג את ה-path אל ה-dataSnapshot שעליו הופעל. לקבלת המחרוזת הפעל עליו שיטת toString().
* hasChildren - בדיקה האם יש צאצאים ל-dataSnpanshot. במידה וכן מחזיר true, אחרת false.
* hasChild - מקבל Path שמתחיל מה-dataSnapshot ומחזיר true אם אכן שמור שם מידע.
* getChildrenCount - מחזיר את מספר הילדים שיש תחת ה-dataSnapshot.
* child - מקבלת צומת שהיא בן ל-dataSnapshot נוכחי ומחזירה dataSnapshot לצומת הבן. לא משנה את ה-dataSnapshot עליו הופעלה השיטה.
* getKey - מחזיר את שם המפתח שעליו מצביע ה-dataSnapshot.
* getValue - מחזיר את הערך ששמור ב-dataSnapshot. במידה ורוצים לשלוף אובייקט מחלקה או string יש להכניס בסוגריים את שם המחלקה, לדוגמה Client.class, String.class. שיטה זו שימושית מאוד עם שיטת child, שכן מאפשרות טיול על העץ ושליפת ערך רצוי.

Client client = dataSnapshot.getValue(Client.class);

* getChildren - מחזיר **רשימה** של כל הצמתים שהם ילדים ישירים של ה-dataSnapshot. בדרך כלל משתמשים בתוך לולאת for each.

For(DataSnapshot data : dataSnapshot.getChildren()) {

Client client = data.getValue(Client.class);

…

}

## שאילתות על רשימה של נתונים

ניתן לבצע שאילתה על DatabaseReference כדי להחזיר רשימה של ערכים, שהם הבנים של ה- DatabaseReferenceעליו הופעלה השיטה, לפי מיון או סינון לפי קריטריונים מסוימים. האובייקט החוזר מכל אחד משיטות אלו הוא אובייקט מסוג Query, שעליו ניתן שוב להפעיל את אחת מן השיטות כדי לדייק את השאילתה. על ה-Query הסופי של השאילתה ניתן להפעיל addValueEventListener, כמו שלמדנו בסעיף קודם, כדי לקרוא את המידע בו.

מיון (תמיד ממיין בסדר עולה):

* orderByChild() – מקבל איזשהו שדה אצל הילדים או path לעומק העץ וממיין לפי תכונה זו.
* orderByKey() – ממיין לפי המפתח של הילדים.
* orderByValue() – ממיין לפי הערך של הילדים.

סינון:

* limitToFirst() – מקבל מספר של הילדים הראשונים שעליו להחזיר.
* limitToLast() – מקבל מספר של הילדים האחרונים שלא יחזיר.
* startAt() – מקבל מחרוזת ומחזיר את כל הילדים שה-key שלהם מתחיל במחרוזת זו.
* endAt() - מקבל מחרוזת ומחזיר את כל הילדים שה-key שלהם מסתיים במחרוזת זו.
* equelTo() – מקבל מחרוזת ומחזיר את כל הילדים שה-key שלהם שווה במחרוזת זו.

דוגמאות:

* מחזיר כל הילדים להם שדה פנימי של name המתחיל בשם chris.

Query query = dbRef.orderByChild("name").equalTo("Chris");//

* מחזיר את שני הילדים שבמיון עולה לפי השדה name הם ראשונים.

Query query = dbRef.orderByChild("name").limitToFirst(2);

## הצגת רשימה של נתונים

אם ב-firebase שמורים רשימה של אובייקטים ורוצים להציג את כולם באקטיביטי, יש לעשות זאת במספר שלבים:

1. יצירת מחלקה המייצגת את האובייקט ששמור ברשימה. לדוגמה Message.
2. ליצור קובץ xml שבו יהיו כל ה-views שנרצה להציג עבור כל אובייקט. שם קובץ זה יהיה custom\_message לדוגמה.
3. ליצור קובץ java שיורש מהמחלקה ArrayAdapter<Message>. בתוך קובץ זה נממש את שיטת getView שבה נגדיר איזו פעולה תתבצע עבור כל אובייקט מהרשימה. בתחילת השיטה ניצור אובייקט view המקבל את קובץ ה-custom\_message שיצרנו בשלב קודם.
4. ניצור קובץ xml לאקטיביטי שבו נציג את כל הרשימה. בתוך קובץ ה-xml נגדיר listView.
5. בקובץ java של האקטיביטי ניצור אובייקט מה-adapter שבנינו בשלב 3 ונכניס אליו את כל רשימת האובייקטים שמשכנו מה-firebase באמצעות ValueEventListener. לסיום ניצור אובייקט ListView המייצג את ה-listView משלב קודם, ונכניס אליו את ה-adapter באמצעות שיטת setAdapter.

\*דוגמה לתהליך כזה בפרויקט Paotonet עבור רשימת הודעות ורשימת ילדים.

ל-listView אפשר להוסיף אפשרות שאם מישהו ילחץ על אחד מהפריטים ברשימה המוצגים באקטיביטי תתבצע איזושהי פעולה. עושים זאת על ידי הפעלת שיטת setOnItemClickListener על אובייקט ה-listView. האובייקט שעליו לחצו מהרשימה יהיה האובייקט במיקום position מרשימת האובייקטים שמשכנו מה-firebase.

list.setAdapter(itemsAdapter);

list.setOnItemClickListener(**new** AdapterView.OnItemClickListener() {

@Override

**public** void onItemClick(AdapterView<?> parent, View view, int position, long id) {

Toast.makeText(MainActivity.**this**, items.get(position)+"", Toast.LENGTH\_SHORT). show();

}

});

## הקשבה לאירועים

אפשר להאזין לאירועים שקורים ברמת הילדים של DatabaseReference כלשהו. יש ליצור אובייקט ChildEventListener שבתוכו נממש שיטה לכל אירוע שאנו רוצים להקשיב לו. אירועים כמו: נוסף ילד, שונה ילד, נמחק ילד או הוזז למקום אחר. את האובייקט הזה יש להפעיל על ה- DatabaseReferenceבאמצעות שיטת addChildEventListener. כאשר לא נרצה יותר להאזין לאירועים כדאי להסיר את ה-listeners באמצעות שיטת removeEventListener.

ChildEventListener childListener = new ChildEventListener() {

@Override

public void onChildAdded(DataSnapshot dataSnapshot, String s) {

}

@Override

public void onChildChanged(DataSnapshot dataSnapshot, String s) {

}

@Override

public void onChildMoved(DataSnapshot dataSnapshot, String s) {

}

@Override

public void onChildRemoved(DataSnapshot dataSnapshot) {

}

};

dbRef.addChildEventListener(childListener);

…

dbRef.removeEventListener(childListener);

## Firebase Authentication

## Rules