AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Broadcast receivers, content providers, services, async tasks

Mateusz Nowotyński **Marcin Moskal** **Kamil Osuch**

12.12.2017

AsyncTask

AsyncTask pozwala na wykonywanie operacji asynchronicznych w interfejsie użytkownika. Wykonuje blokujące operacje w wątku roboczym, a następnie publiuje rezultaty w wątku UI, nie wymagając od użytkownika zarządzania wątkami. AsyncTask możemy wywołać tylko raz.

Aby użyć AsyncTask należy stworzyć podklasę AsyncTask i zaimplementować metodę **doInBackground**. Aby po wykonaniu operacji dokonać update'u UI, trzeba zaimplementowac metodę **onPostExecute**, która odbiera rezultat z doInBackground i wykonuje się na wątku UI.

Wystartowanie AsyncTask

```
asyncTask.execute()
```

AsyncTask - kroki wykonania

Gdy AsyncTask jest wywoływany, przechodzi przez 4 kroki:

1. **onPreExecute()**
2. **doInBackground()**
3. **onProgressUpdate()**
4. **onPostExecute()**

AsyncTask - kroki wykonania

- **onPreExecute()** - wywoływany na wątku UI przed wystartowaniem task'a. Zwykle wykorzystywany do przygotowania task'a (np. poprzez pokazanie pasku ładowania)
- **doInBackground()** - wywoływany na wątku roboczym, natychmiast po onPreExecute. Tutaj znajdują się wszystkie operacje, które mogą trwać długo. Ta metoda może użyć metody **publishProgress**, aby przekazać informację o postępie do metody onProgressUpdate

AsyncTask - kroki wykonania

- **onProgressUpdate()** -wywołany na wątku UI po wywołaniu publishProgress. Czas wykonania jest nieokreślony. Metoda ta jest wykorzystywana do wyświetlania jakiejś formy postępu w interfejsie użytkownika.
- **onPostExecute()** - wywołany na wątku UI po zakończeniu operacji w tle. Jest tutaj przekazany rezultat z metody doInBackground

AsyncTask - anulowanie

Task może być w każdej chwili anulowany poprzez wywołanie metody **cancel**. Po wywołaniu tej metody zamiast wykonania onPostExecute po zakończeniu operacji w tle, zostanie wywołana metoda **onCancelled**. Dodatkowo wszystkie wywołania metody **isCancelled** będą zwracały true, co pozwoli na szybsze anulowanie task'a (np. poprzez sprawdzenie w doInBackground czy task nie został anulowany).

AsyncTask - przykład

```
public class Obliczenia extends AsyncTask<Void, Void, Void> {

    WeakReference<Activity> activity;

    public Obliczenia(Activity view) {
        this.activity = new WeakReference<>(view)
    }

    @Override
    protected void onPreExecute() {
        activity.showDialog(MainActivity.PLEASE_WAIT_DIALOG);
    }

    @Override
    protected Void doInBackground(Void... arg0) {
        try {
            Thread.sleep( millis: 5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return null;

    }

    @Override
    protected void onPostExecute(Void result) {
        Activity view = this.activity.get();
        view.removeDialog(MainActivity.PLEASE_WAIT_DIALOG);
        Toast.makeText(view,  text: "Obliczono!", Toast.LENGTH_SHORT).show();
    }

}
```

Serwisy

Serwis jest komponentem aplikacji, który wykonuje długo trwające operacje (np: update do bazy danych, pobieranie pliku z sieci) oraz nie zapewnia UI(user interface).
Serwisy są uruchamiane przez inne komponenty np: Activity.
Dodatkowo serwis może zostać "związany" z komponentem który go uruchomił, w takim przypadku możemy wchodzić w interakcje z serwisem czyli wykonywać metody które się w nim znajdują.

Typy serwisów

- **Foreground** - wykonuje operacje zauważalne dla użytkownika. Serwisy te działają nawet wtedy gdy użytkownik nie uzywa aplikacji
- **Background** - wykonuje operacje, o których użytkownik nie jest bezpośrednio informowany.
- **Bound** - Serwis jest związany gdy komponent aplikacji wiąże się z nim przy pomocy metody **bindService**. Komponent dostaje wtedy do dyspozycji prosty interface IBinder dzięki któremu może nawiązać komunikacje z serwisem.

Tworzenie serwisu

Aby stworzyć serwis, trzeba stworzyć podklasę klasy Service lub jednej z jej istniejących już podklas. Metody które należy przeciążyć podczas tworzenia serwisu:

- **onStartCommand()** - metoda wywoływana po wywołaniu startService. Po wykonaniu tej metody serwis jest wystartowany i może działać w tle przez czas nieokreślony. Jeśli przeciążymy tą metodę, to zatrzymanie serwisu trzeba takze zaimplementować poprzez wywołanie metod **stopSelf** lub **stopService**.
- **onBind()** - metoda wywoływana po wywołaniu bindService. Musi się tu znaleźć implementacja interfejsu którego klient będzie używać do komunikacji z serwisem.

Tworzenie serwisu

- **onCreate()**
- **onDestoy()**

Dodanie serwisu do manifestu

Wszystkie serwisy muszą być zadeklarowane w manifeście aplikacji.

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".ExampleService" />
      ...
  </application>
</manifest>
```

Przykład implementacji started service

```
public class ProstySerwis extends IntentService{
    private int i;
    private Handler handler = new Handler();
    public ProstySerwis() {
        super("ProstySerwis");
    }

    @Override
    protected void onHandleIntent(Intent arg0) {

        while(i<10){
            try {
                Thread.sleep( millis: 5000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            handler.post(new Runnable() {
                public void run() {
                    Toast.makeText(getApplicationContext(),
                            text: "WITAJ W SERWISIE", Toast.LENGTH_SHORT).show();
                }
            });
            i++;
        }
    }

}
```

Przykład wystartowania started service

```
startService(new Intent(getBaseContext(), ProstySerwis.class));
```

Przykład implementacji bound service

```
public class BoundService extends Service {
    private Handler handler = new Handler();

    private final IBinder mBinder = new LocalBinder();

    public class LocalBinder extends Binder {
        BoundService getService() {
            //zwracamy instancje serwisu, przez nią odwołamy się następnie do metod.
            return BoundService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }
    //metoda którą zapewniamy.
    public void generateToast() {
        handler.post(new Runnable() {
            public void run() {
                Toast.makeText(getApplicationContext(),
                        text: "WITAJ W SERWISIE(znowu)", Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

Przykład połączenia z bound service

```
Intent intent = new Intent(this, BoundService.class);
bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
```

Broadcast Receivers

Do czego to służy?

BroadcastReceiver pozwala nam na odbieranie powiadomień (Systemu bądź innej aplikacji) wewnątrz naszej aplikacji. Takim powiadomieniem może być na przykład informacja o nowej wiadomości SMS bądź rozładowanej baterii.

Tworzenie Broadcast Receiver'a

Żeby zbudować nasz własny BroadcastReceiver musimy wykonać dwie czynności:

1. Stworzyć podklasę klasy BroadcastReceiver
2. Wyspecyfikować receiver w manifeście aplikacji lub bezpośrednio w kodzie

Przykład klasy Broadcast Receiver'a

```
public class MyBroadcastReceiver extends BroadcastReceiver {
    private static final String TAG = "com.example.woy.toolbar.MyBroadcastReceiver";

    @Override
    public void onReceive(Context context, Intent intent) {
        StringBuilder sb = new StringBuilder();
        sb.append("Action: " + intent.getAction() + "\n");
        sb.append("URI: " + intent.toUri(Intent.URI_INTENT_SCHEME).toString() + "\n");
        String log = sb.toString();
        Log.d(TAG, log);
        Toast.makeText(context, log, Toast.LENGTH_LONG).show();
    }
}
```

Dodanie do manifestu

```
<receiver android:name=".MyBroadcastReceiver"  android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
        <action android:name="android.intent.action.INPUT_METHOD_CHANGED" />
    </intent-filter>
</receiver>
```

Uwaga!

Powyższe rozwiązanie nie zadziała w wersji API wyższej niż 25. Nie można wtedy użyć manifestu do zadeklarowania receiver'a dla większości implicit broadcast'ów (z wyjątkiem kilku wyszczególnionych).

Rejestracja w kodzie

```
public class MainActivity extends Activity {

    private IntentFilter filter =
            new IntentFilter( action: "android.provider.Telephony.SMS_RECEIVED");

    private BroadcastReceiver broadcast = new MyBroadcastReceiver();

    @Override
    public void onResume() {
        super.onResume();
        registerReceiver(broadcast, filter);
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public void onPause() {
        unregisterReceiver(broadcast);
        super.onPause();
    }
}
```

Rejestracja - kontekst

Manifest

W przypadku dodania deklaracji receiver'a do manifestu, jest on rejestrowany w momencie zainstalowania aplikacji. Receiver staje się wtedy oddzielnym punktem wejścia aplikacji. Oznacza to, że system może wystartować aplikację i przekazać wysyłany broadcast do receiver'a.

Rejestracja w kodzie

Receiver'y zarejestrowane w kodzie odbierają broadcast'y tak długo jak kontekst w którym zostały zarejestrowane jest aktywny.

Wysyłanie broadcast'ów

Android oferuje trzy sposoby wysyłania broadcastów:

- **sendOrderedBroadcast** - wysyła broadcast do jednego receiver'a na raz. Gdy receiver zakończy swoje wykonanie, może propagowac rezultat do innego receiver'a, lub też całkowicie przerwać broadcast. Kolejność broadcastów może być określana za pomocą atrybutu **android-priority**

Wysyłanie broadcast'ów

Android oferuje trzy sposoby wysyłania broadcastów:

- **sendOrderedBroadcast** - wysyła broadcast do jednego receiver'a na raz. Gdy receiver zakończy swoje wykonanie, może propagowac rezultat do innego receiver'a, lub też całkowicie przerwać broadcast. Kolejność broadcastów może być określana za pomocą atrybutu **android-priority**
- **sendBroadcast** - tzw. normalny broadcast, wysyła broadcast do wszystkich receiver'ów w nieokreślonej kolejności. Jest szybszy, ale nie można kontrolować przeplywu broadcast'u między receiver'ami, czy go zatrzymać.

Wysyłanie broadcast'ów

Android oferuje trzy sposoby wysyłania broadcastów:

- **sendOrderedBroadcast** - wysyła broadcast do jednego receiver'a na raz. Gdy receiver zakończy swoje wykonanie, może propagowac rezultat do innego receiver'a, lub też całkowicie przerwać broadcast. Kolejność broadcastów może być określana za pomocą atrybutu **android-priority**
- **sendBroadcast** - tzw. normalny broadcast, wysyła broadcast do wszystkich receiver'ów w nieokreślonej kolejności. Jest szybszy, ale nie można kontrolować przeplywu broadcast'u między receiver'ami, czy go zatrzymać.
- **LocalBroadcastManager.sendBroadcast** - wysyła broadcast do receiver'ów, które są w tej samej aplikacji, co strona wysyłająca.

Przykład wysyłania broadcast'ów

```
Intent intent = new Intent();
intent.setAction("com.example.broadcast.MY_NOTIFICATION");
intent.putExtra( name: "data", value: "Example data");
sendBroadcast(intent);
```

Uwaga!

Pomimo tego, że Intent jest używany zarówno do wysyłania broadcast'ów, jak i startowania activity (przy pomocy **startActivity**), akcje te nie są ze sobą powiązane. Broadcast receiver'y nie odbiorą akcji używanej do wystartowania activity.

Ograniczenia broadcast'ów - uprawnienia

Wysyłanie

sendBroadcast(intent, permission) - wysłany w ten sposób broadcast może zostać odebrany tylko przez receiver posiadający uprawnienie **permission**

Odbieranie

registerBroadcaster(receiver, intent-filter, permission, handler) - Receiver odbierze broadcast'y tylko od wysyłających, którzy posiadają uprawnienie **permission**

Content providers

Content provider pomaga aplikacji w zarządzaniu dostępem do danych przechowywanych przez samą siebie czy inne aplikacje, i udostępnia sposób na dzielenie się danymi z innymi aplikacjami.

Content provider prezentuje dane zewnętrznym aplikacjom jako jedna lub wiele tabel, które są podobne do tabel z relacyjnych baz danych. Wiersz reprezentuje instancję jakiegoś typu danych, który gromadzi w sobie provider. Kolumna reprezentuje indywidualną część danych dla konkretnej instancji.

Dostęp do danych

Do uzyskania dostępu do danych wykorzystywany jest **ContentResolver**. Każdy kontekst aplikacji przechowuje instancję klasy ContentResolver, do której dostęp obywa się poprzez metodę **getContentResolver()**.

Metody operujące na danych

Metody klasy ContentResolver udostępniają podstawowe operacje CRUD(create - **insert()**, read - **query()**, update - **update()**, delete - **delete()**). Metody te jako jeden z argumentów przyjmują adres URI, na którego podstawie ContentResolver decyduje z którego providera skorzystać.

Content URI

Schemat URI

content://<authority>/<data-type>/<id>

- **authority** - nazwa symboliczna content provider'a
- **data-type** - typ danych które oferuje dany provider
- **id** - numer konkretnego rekordu zapisanego w providerze

Przykładowe zapytanie o dane

```
mCursor = getContentResolver().query(
        UserDictionary.Words.CONTENT_URI,   // The content provider URI
        mProjection,                        // The columns to return for each row
        mSelectionClause,                    // Selection criteria
        mSelectionArgs,                     // Selection criteria
        mSortOrder);                        // The sort order for the returned rows
```

Zapytania powinny być wykonywane na innym wątku niż wątek UI, asynchronicznie. Jednym ze sposobów jest użycie klasy **CursorLoader**.

Aby móc pobierać dane z provider'a, aplikacja musi posiadać uprawnienia do odczytu z provider'a.

Argumenty query()

`Uri`	`FROM table_name`	`Uri` maps to the table in the provider named *table_name*.
`projection`	`col,col,col,...`	`projection` is an array of columns that should be included for each row retrieved.
`selection`	`WHERE col = value`	`selection` specifies the criteria for selecting rows.
`selectionArgs`	(No exact equivalent. Selection arguments replace `?` placeholders in the selection clause.)	
`sortOrder`	`ORDER BY col,col,...`	`sortOrder` specifies the order in which rows appear in the returned `Cursor`.

Wyświetlanie rezultatów

Funkcja query zwraca zawsze obiekt klasy **Cursor**, który udostępnia losowy dostęp do wierszy i kolumn które zawiera. Dane te można następnie przekonwertować na ListView przy pomocy klasy **SimpleCursorAdapter**, lub użyć w innych miejscach (Cursor ma kilka metod get służących do pobierania różnych typów danych z obiektu).

Wstawianie danych

```
Uri mNewUri;
ContentValues mNewValues = new ContentValues();

mNewValues.put(UserDictionary.Words.APP_ID, "example.user");
mNewValues.put(UserDictionary.Words.LOCALE, "en_US");
mNewValues.put(UserDictionary.Words.WORD, "insert");
mNewValues.put(UserDictionary.Words.FREQUENCY, "100");

mNewUri = getContentResolver().insert(
        UserDictionary.Word.CONTENT_URI,
        mNewValues
);
```

Dane dla nowego wiersza tworzymy przy pomocy klasy **ContentValues**, która reprezentuje pojedynczy wpis. Nie dodaje się kolumny ID, która jest dodawana automatycznie. Metoda insert zwraca content URI do wstawianego obiektu.

Aktualizacja danych

Do aktualizacji danych także używana jest klasa ContentValues, zawierająca tylko wartości, które chcemy zaktualizować. W metodzie update podawane są argumenty, które odpowiadają za wyszukanie wierszy, które mają zostać zaktualizowane. Metoda zwraca ilość zaktualizowanych wierszy.

Aktualizacja danych

```
ContentValues mUpdateValues = new ContentValues();

String mSelectionClause = UserDictionary.Words.LOCALE +  "LIKE ?";
String[] mSelectionArgs = {"en_%"};

int mRowsUpdated = 0;

mUpdateValues.putNull(UserDictionary.Words.LOCALE);

mRowsUpdated = getContentResolver().update(
        UserDictionary.Words.CONTENT_URI,
        mUpdateValues
        mSelectionClause
        mSelectionArgs
);
```

Usuwanie danych

```
String mSelectionClause = UserDictionary.Words.APP_ID + " LIKE ?";
String[] mSelectionArgs = {"user"};

int mRowsDeleted = 0;

mRowsDeleted = getContentResolver().delete(
        UserDictionary.Words.CONTENT_URI,
        mSelectionClause
        mSelectionArgs
);
```

Do metody delete podajemy argumenty, które wyszukują wiersze, które chcemy usunąć. Metoda zwraca ilość usuniętych wierszy.

Tworzenie Content Provider'a

Tworzenie Content Provider'a polega na zaimplementowaniu metod klasy ContentProvider. Klasa ta definiuje 6 metod abstrakcyjnych które wymagają implementacji:

- query() - pobiera dane z provider'a, zwraca obiekt klasy Cursor
- insert() - wstawia dane do provider'a, zwraca Uri do wstawionego obiektu
- update() - aktualizuje wybrane wiersze, zwraca ilość zaktualizowanych wierszy
- delete() - usuwa wybrane wiersze, zwraca ilość usuniętych wierszy
- getType() - zwraca MIME type odpowiadający podanemu URI
- onCreate() - inicjalizuje provider

Tworzenie Content Provider'a - kroki

1. Stworzenie podklasy klasy ContentProvider
2. Zdefiniowanie URI którym będzie posługiwał się Content Provider
3. Stworzenie bazdy danych
4. Implementacja różnych query wykonujących różne operacje na bazie danych
5. Zarejestrowanie provider'a w manifeście poprzez użycie tagu **<provider>**