MOBILNE APLIKACIJE

Vežbe 5

Servisi i prijemnici poruka

2024/2025

Sadržaj

1. Servisi

Tokom programiranja, često ćete imati priliku da se susretnete sa pojmom *servis.* Šta on tačno podrazumeva u kontekstu Androida? Pod servisom podrazumevamo operacije koje ne zahtevaju interakciju korisnika, već se izvršavaju u pozadini. To je komponenta koja izvršava duge operacije u pozadini i služi za implementaciju klijent-server arhitekture. Koristi se za izvršavanje akcija kao što su: puštanje muzike, rad sa učitavanjem i ispisom fajlova, preuzivanje fajlova, itd.

Postoje 3 vrste servisa:

- *Background* servis
- *Foreground* servis
- *Bound* servis

Background servis se pokreće samo kada je aplikacija pokrenuta, tako da će biti prekinut kada se aplikacija prekine. Izvršava se neodređeno vreme u pozadini i zaustavlja se kada završi operaciju. Neophodno je da bude eksplicitno kreiran od strane aktivnosti ili fragmenta koji ga poziva. Ne pruža korisnicima nikakvu informaciju o tome kakvu operaciju izvršava.

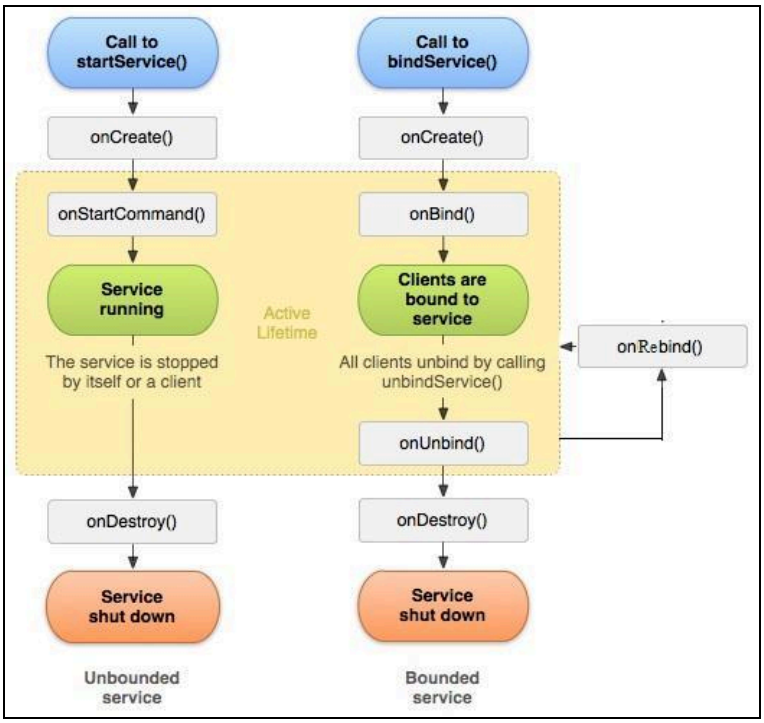
Za razliku od *Background* servisa, *Foreground* servis je servis koji ostaje živ čak i kada je aplikacija prekinuta. Korisnicima pruža informaciju o izvršavanoj operaciji putem notifikacija. Notifikacije treba da se prikazuju i kada aplikacija nije više u fokusu.

Bound servis ili vezan servis, se izvršava samo dok je komponenta za koju je vezan i dalje aktivna. On nudi interfejs koji omogućava komponentama da komuniciraju sa njim (šalju se zahtevi i dobijaju se odgovori).

1.1 Životni ciklus servisa

Isto kao što aktivnosti imaju svoj životni ciklus, tako i servisi imaju svoj. Životni ciklus servisa je jednostavniji od životnog ciklusa aktivnosti. On poseduje različite metode koje se pozivaju prilikom prelaska iz jednog u drugo stanje. Na slici 1 možemo da vidimo životni ciklus servisa.

Početak životnog veka je od metode *onCreate*, a završetak je sa metodom *onDestroy*. Servis je aktivan (aktivni životni vek) od metode *onStartCommand* ili *onBind,* pa sve do metode *onDestroy* ili *onUnbind*.



Slika 1. Životni ciklus servisa

OnCreate

Ova metoda se poziva nakon što neka komponenta zatraži pravljenje servisa. Ovde navodimo samu svrhu servisa.

onStartCommand

Kada neka komponenta želi da kreira servis, ona pozove metodu *startService.* U tom trenutku sistem će pozvati metodu *onStartCommand*.

onBind

Kada neka komponenta zatraži da se napravi vezani servis ona poziva metodu *bindService*. Nakon metode *bindService* poziva se metoda *onBind*.

onUnbind

Metoda *onUnbind* se poziva posle poziva *unbindService* metode.

onRebind

onRebind pozivamo posle poziva *bindService*, ako je prethodno izvršena *onUnbind* metoda.

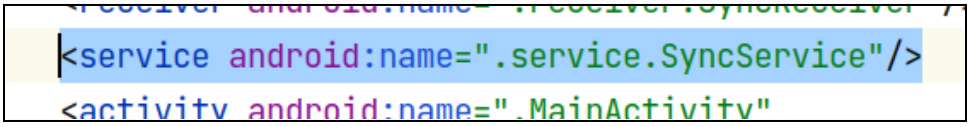
onDestroy

Na samom kraju, pozivamo metodu *onDestroy* prilikom uništavanja servisa.

Servis može da se zaustavi pozivom metode *stopSelf*, može neka druga komponenta da ga zaustavi pozivom metode *stopService* ili ga zaustavlja Android da bi oslobodio memoriju.

1.2 Pravljenje *background* servisa

Da bismo što bolje razumeli šta je servis, napravićemo ga. Pravimo novu klasu sa nazivom *SyncService* koja nasleđuje klasu *Service*. Po uzoru na aktivnosti, servis registrujemo u *AndroidManifest* datoteci (slika 2).



Slika 2. Navođenje servisa u AndroidManifest-u

Servis poseduje metodu *onStartCommand* koja se poziva prilikom izvršavanja zadatka servisa (slika 3). U našem primeru servis će, ako su zadovoljeni uslovi, pokrenuti asinhroni zadatak.

U klasi *CheckConnectionTools* kreirali smo pomoćnu metodu, koja poziva metode Android operativnog sistema i proverava da li je uređaj povezan na internet, i ako jeste na koji je to način povezan. Ta pomoćna metoda kao rezultat vraća jedan od brojeva 0, 1 ili 2, u zavisnosti od toga da li je uređaj povezan na wifi, da li koristi mobilne podatke ili nije povezan na internet. Ovu pomoćnu metodu pozivamo u servisu i povratnu vrednost čuvamo u *status-u* (linija 31).
U slučaju da je uslov zadovoljen, tj. da je uređaj povezan na internet, pokrećemo asinhroni zadatak (linija 34).
Na kraju (linija 52) pozivamo metodu *stopSelf*, koja će zaustaviti servis.

```
29  public int onStartCommand(Intent intent, int flags, int startId) {
30      Log.i( tag: "REZ",  msg: "SyncService onStartCommand");
31      int status = CheckConnectionTools.getConnectivityStatus(getApplicationContext());
32      if(status == CheckConnectionTools.TYPE_WIFI || status == CheckConnectionTools.TYPE_MOBILE){
33          // Alternativa za SyncTask
34          executor.execute(() -> {
35              //Background work here
36              Log.i( tag: "REZ",  msg: "Background work here");
37              try {
38                  Thread.sleep( millis: 1000);
39              } catch (InterruptedException e) {
40                  e.printStackTrace();
41              }
42              handler.post(() -> {
43                  //UI Thread work here
44                  Log.i( tag: "REZ",  msg: "UI Thread work here");
45                  Intent ints = new Intent(HomeActivity.SYNC_DATA);
46                  int intsStatus = CheckConnectionTools.getConnectivityStatus(getApplicationContext());
47                  ints.putExtra(RESULT_CODE, intsStatus);
48                  getApplicationContext().sendBroadcast(ints);
49              });
50          });
51      }
52      stopSelf();
53      return START_NOT_STICKY;
54  }
```

Slika 3. Metoda servisa *onStartCommand*

1.2 Pravljenje *foreground* servisa

Za kreiranje *foreground* servisa, deklarišemo komponentu u *AndroidManifest.xml* i navodimo tip željenog servisa. U našem slučaju, kreiraćemo servis koji omogućava reprodukciju zvuka (slika 4).

```
<service
    android:name=".services.ForegroundService"
    android:foregroundServiceType="mediaPlayback"
    android:enabled="true"
    android:exported="false" />
```

Slika 4. Deklarisanje *foreground* servisa za reprodukciju zvuka

Servis se pokreće/zaustavlja klikom na *switch* komponentu unutar *SettingsFragment*-a, prilikom čega se kreiraju namere u kojima navodimo naziv servisa. Namere, zatim, prosleđujemo *startForegroundService()* i *stopService()* metodama (slika 5).

```
startServiceButton.setOnCheckedChangeListener((compoundButton, b) -> {
    Intent intent = new Intent(getActivity(), ForegroundService.class);
    intent.setAction(ForegroundService.ACTION_START_FOREGROUND_SERVICE);
    if (b) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            requireActivity().startForegroundService(intent);
        } else {
            requireActivity().startService(intent);
        }
    } else {
        Intent intent1 = new Intent(getActivity(), ForegroundService.class);
        intent.setAction(ForegroundService.ACTION_STOP_FOREGROUND_SERVICE);
        requireActivity().stopService(intent1);
    }
});
```

Slika 5. Pokretanje i zaustavljanje servisa

Sam servis treba da pokrene plejer za reprodukciju zvuka i da omogući pauziranje i zaustavljanje reprodukcije. Ovo nam omogućava *startForegroundService()* metoda unutar *ForegroundService* klase (slika 6.). Osim pokretanja plejera, metoda se brine i o definisanju notifikacija koje su neophodne za rad sa *Foreground* servisima o kojima će biti više reči u narednom odeljku.

```
player = MediaPlayer.create( context: this, Settings.System.DEFAULT_RINGTONE_URI);
player.setLooping(true);
player.start();
```

Slika 6. Pokretanje plejera

Unutar *onStartCommand()* metode, u zavisnosti od prosleđene akcije unutar namere, okidamo različite operacije (slika 7).

ACTION_START_FOREGROUND_SERVICE poziva gore pomenutu metodu i pokreće reprodukovanje zvuka. Korisniku se prikazuje notifikacija sa mogućnostima klika na *PLAY, PAUSE* i *STOP*.

ACTION_PLAY omogućava ponovno pokretanje zvuka nakon pauziranja ili stopiranja (klikom na *PAUSE* ili *STOP* preko notifikacije).

ACTION_PAUSE pauzira zvuk, dok *ACTION_STOP* stopira reprodukovanje i zaustavlja *foreground* servis nakon čega se uklanja i notifikacija.

```
String action = intent.getAction();
Log.i( tag: "SERVICE STARTED",  msg: "YES");
switch (action)
{
    case ACTION_START_FOREGROUND_SERVICE:
        startForegroundService();
        Toast.makeText(getApplicationContext(),  text: "Foreground service is started.", Toast.LENGTH_LONG).show();
        break;
    case ACTION_PLAY:
        player.start();
        Toast.makeText(getApplicationContext(),  text: "You clicked Play button.", Toast.LENGTH_LONG).show();
        break;
    case ACTION_PAUSE:
        Toast.makeText(getApplicationContext(),  text: "You clicked Pause button.", Toast.LENGTH_LONG).show();
        player.pause();
        break;
    case ACTION_STOP:
        Toast.makeText(getApplicationContext(),  text: "You clicked Stop button.", Toast.LENGTH_LONG).show();
        player.stop();
        stopForeground( removeNotification: true);
        break;
}
```

Slika 7. Pokretanje različitih operacija unutar servisa

2. Notifikacije

Notifikacije su poruke koje Android prikazuje korisniku mimo korisničkog interfejsa same aplikacije. Služe za obaveštavanje korisnika o događajima unutar samih aplikacija, prikaz podsetnika, itd. Obično se prikazuju u gornjem delu ekrana uređaja gde je moguće proširiti, ukloniti, reagovati ili izvršiti određenu akciju nad njima, kao i pokrenuti aplikaciju od koje je stiglo obaveštenje.

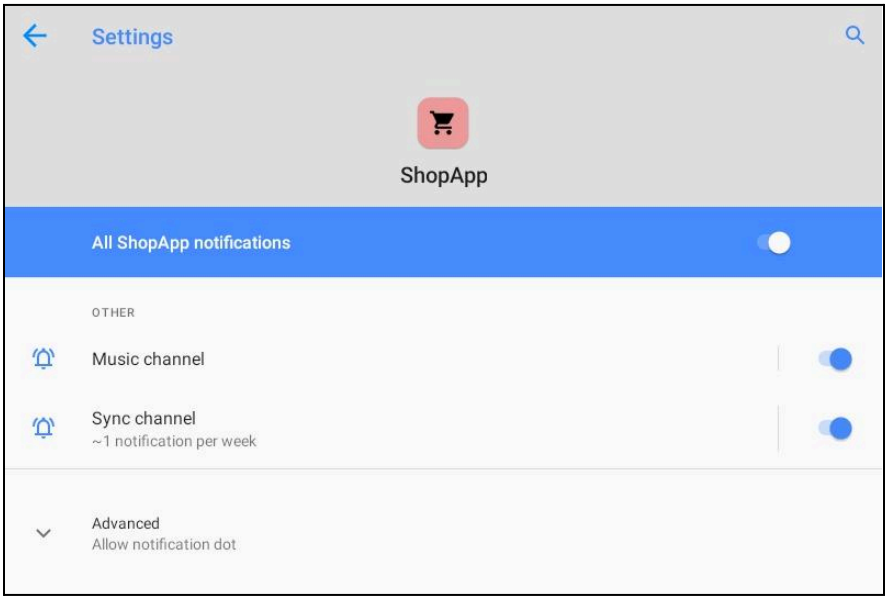
2.1 Kreiranje notifikacija

Da bi se notifikacije mogle prikazivati, neophodno je kreirati kanal putem kojih će pristizati. Kanal za slanje i prijem notifikacija sadrži ime, opis i definisan prioritet notifikacija u zavisnosti od kod se određuje način prikazivanja samih poruka. Kreiranje kanala prikazano je na slici 8.

```
private void createNotificationChannel(CharSequence name, String description, String channel_id, int importance) {
    // Create the NotificationChannel, but only on API 26+ because
    // the NotificationChannel class is new and not in the support library
    NotificationChannel channel = new NotificationChannel(channel_id, name, importance);
    channel.setDescription(description);
    // Register the channel with the system; you can't change the importance
    // or other notification behaviors after this
    NotificationManager notificationManager = getSystemService(NotificationManager.class);
    notificationManager.createNotificationChannel(channel);
}
```

Slika 8. Kreiranje notifikacionog kanala

U aplikaciji je moguće imati više kanala što korisniku omogućava kontrolu nad vizuelnim i audio podešavanjima notifikacija koje pripadaju jednoj grupi. Unutar naše aplikacije kreirana su dva notifikaciona kanala: MUSIC_CHANNEL_ID i SYNC_CHANNEL_ID. Korisnik može omogućiti i onemogućiti prikaz svih notifikacija koje su vezane za ove kanale posebno (slika 9) kroz sistemska podešavanja.



Slika 9. Podešavanja prikaza notifikacija

2.2 Kreiranje notifikacija

Kao što je već napomenuto, unutar *startForegroundService()* metode, definiše se notifikacija koja će se prikazati korisniku nakon pokretanja plejera (slika 10).

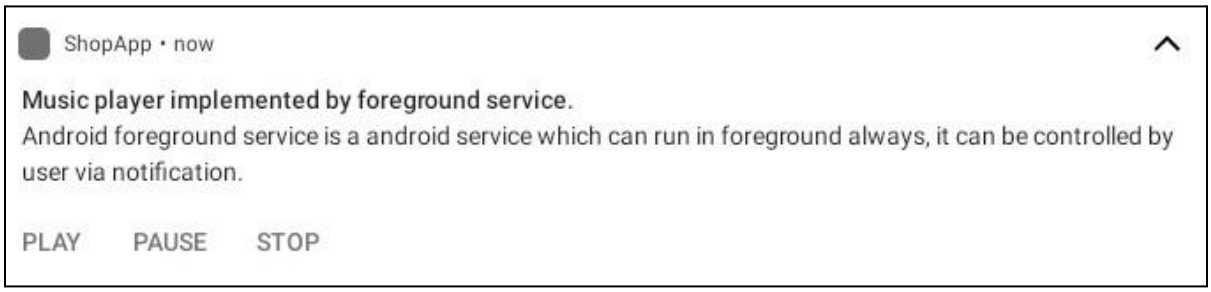
Za sam prikaz notifikacije neophodno je kreirati nameru (intent) i preuzeti kanal putem kog će notifikacija biti prikazana (linije 94-98). Notifikaciji se može dodeliti font, veličina slova, proslediti tekst, podesiti ikonica i veličina ikonice, kao i prioritet (linije 101-114).

Notifikacijama se mogu podesiti i akcije putem kojih korisnik može da odreaguje na pristiglo obaveštenje. Linije 115-120 prikazuju definisanje *PLAY* akcije. Kreira se namera kojoj se postavlja akcija *ACTION_PLAY.* Unutar prethodno pomenute *onStartCommand()* metode *foreground* servisa (slika 7) očekuje se pristizanje ovakve akcije i reaguje se pokretanjem plejera.

```
94      Intent intent = new Intent();
95      PendingIntent pendingIntent = PendingIntent.getActivity( context: this, notificationID, intent, PendingIntent.FLAG_MUTABLE);
96
97      // Create notification builder.
98      NotificationCompat.Builder builder = new NotificationCompat.Builder( context: this, MUSIC_CHANNEL_ID);
99
100     // Make notification show big text.
101     NotificationCompat.BigTextStyle bigTextStyle = new NotificationCompat.BigTextStyle();
102     bigTextStyle.setBigContentTitle("Music player implemented by foreground service.");
103     bigTextStyle.bigText( cs: "Android foreground service is a android service which can run in foreground always," +
104             "it can be controlled by user via notification.");
105     // Set big text style.
106     builder.setStyle(bigTextStyle);
107
108     builder.setWhen(System.currentTimeMillis());
109     builder.setSmallIcon(R.mipmap.ic_launcher);
110     Bitmap largeIconBitmap = BitmapFactory.decodeResource(getResources(), R.drawable.ic_music);
111     builder.setLargeIcon(largeIconBitmap);
112     // Make head-up notification.
113     builder.setFullScreenIntent(pendingIntent,  highPriority: true);
114
115     // Add Play button intent in notification.
116     Intent playIntent = new Intent( packageContext: this, ForegroundService.class);
117     playIntent.setAction(ACTION_PLAY);
118     PendingIntent pendingPlayIntent = PendingIntent.getService( context: this,  requestCode: 0, playIntent, PendingIntent.FLAG_MUTABLE);
119     NotificationCompat.Action playAction = new NotificationCompat.Action(android.R.drawable.ic_media_play,  title: "Play", pendingPlayIntent);
120     builder.addAction(playAction);
```

Slika 10. Kreiranje notifikacije

Na slici 11 prikazana je ovako kreirana notifikacija.



Slika 11. Notifikacija sa definisanim akcijama

3. Asinhroni zadaci

Android poseduje glavnu nit (*MainThread*) koja crta korisnički interfejs i odgovara na interakciju korisnika. Da ne bismo došli u situaciju da blokiramo glavnu nit i time blokiramo UI, radnje koje se sporo izvršavaju smeštamo na drugu nit.

Glavna nit je jedina koja može da ažurira UI, te je potrebno da se sa podacima, iz radnji koje su bile duge, vratimo na nju.

Asinhrone zadatke koristimo da bismo olakšali asinhrono izvršavanje operacija. Asinhroni zadaci automatski izvršavaju blokirajuću operaciju u sporednoj, pozadinskoj niti i vraćaju rezultat UI niti. Svi asinhroni zadaci jedne aplikacije izvršavaju se u jednoj niti.

3.1 Pravljenje asinhronog zadatka

Kod starijih verzija API level-a (< 30) koristila se klasa *AsyncTask* za kreiranje asinhronih zadataka. Nakon nasleđivanja klase *AsyncTask*, mogli smo da redefinišemo i primenimo metode *onPreExecute, doInBackground, onProgressUpdate i onPostExecute.*

Od verzije API level (>= 30) ova klasa se više ne koristi i postoji mogućnost kreiranja asinhronog zadatka pomoću *ExecutorService* klase.

Potrebno je prvo napraviti instancu *ExecutorService-a* i kreirati nit (*thread*) pomoću metode *newSingleThreadExecutor().* Primer dat na slici 12.

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

Slika 12. Metoda *newSingleThreadExecutor()*

Pomoću instance *executor* moguće je izvršiti asinhron zadatak pomoću metode *execute()* u okviru koje je moguće implementirati *Runnable* funkcionalni interfejs koji je namenjen za pokretanje niti i definiše metod *run()* koji se poziva bez argumenata u okviru pozadinske niti (background), gde se izvršava sav posao koji dugo traje (slika 13).

```
executorService.execute(new Runnable() {
    @Override
    public void run() {
        Log.i( tag: "REZ",  msg: "Background work here");


    }
});
```

Slika 13. Metoda *run() u background-u*

Kako bismo pokrenuli određene procese na glavnoj UI niti, moguće je unutar pozadinske niti pozvati metodu *runOnUiThread()* u okviru koje je takođe moguće implementirati *Runnable* interfejs i definisati metodu *run()* koja se sada pokreće u okviru glavne UI niti (slika 14). Ovu metodu je moguće pozvati samo iz aktivnosti.

```
executorService.execute(new Runnable() {
    @Override
    public void run() {
        Log.i( tag: "REZ",  msg: "Background work here");
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                Log.i( tag: "REZ",  msg: "UI Thread work here");
            }
        });
```

Slika 14. Metoda *runOnUiThread()*

Da biste izvršili operaciju na UI threadu iz servisa, možete koristiti Handler vezan za glavni looper (Looper.getMainLooper()). To omogućava slanje poruke ili pokretanje Runnable zadatka na UI threadu, što je ključno za izmene korisničkog interfejsa ili druge operacije koje moraju biti izvršene na UI threadu.

3.2 Handler i Looper

Ako želimo da ponovo koristimo nit (na primer: da izbegnemo stvaranje novih niti i smanjimo svoj memorijski prostor), moramo da je zadržimo u životu i da nekako osluškuje nova uputstva. Uobičajeni način da se to postigne jeste kreiranje petlje (*Looper*), u ovom primeru, unutar niti *run()* metode. *Looper* održava svoju nit živom, u ovom primeru glavnu nit. Niti podrazumevano nemaju vezan *Looper* za sebe, ali ga možemo kreirati. Može postojati samo jedan *Looper* po niti.

Da bismo mogli da koristimo *Looper* i pokrenemo zadatke unutar UI niti potreban nam je *Handler*. *Handler* je odgovoran za dodavanje poruka u red *Looper*-a, a kada dođe njihovo vreme, odgovoran je za izvršavanje istih poruka na *Looper*-ovoj niti. Kada se kreira *Handler* on je usmeren ka određenom *Looper-u* tj. ka određenoj niti. Metoda *post()* kreira poruku i dodaje je na kraj reda *Looper*-a. Možemo kreirati *Handler* koji je vezan za određeni *Looper* na sledeći način dat na slici 15.

Ovaj pristup omogućava komunikaciju između pozadinskog servisa i UI thread-a bez rizika od izazivanja izuzetaka zbog nepravilnog pristupa korisničkom interfejsu iz pozadinskog thread-a.

Slika 15. Kreiranje *handler*-a i metoda *post()*

4. Prijemnici poruka

Prijemnik poruka (*BroadcastReceiver*) je komponenta koja obrađuje i odgovara na poruke, koje dobije od drugih aplikacija ili samog sistema. Sistem šalje poruke kada je npr. baterija prazna, ekran isključen itd. Aplikacije emituju obaveštenja kada su npr. neki podaci uspešno skinuti sa interneta.

Prijemnici poruka se često koriste u sprezi sa servisima i asinhronim zadacima i najčešće samo obaveštavaju druge komponente da počnu sa izvršavanjem određenih zadataka.

Prijemnici poruka mogu da obrađuju 2 vrste događaja:

- *Normalni događaji* – asinhroni su; prijemnici ih obrađuju nedefinisanim redosledom; efikasniji su.
- *Uređeni događaji* – obrađuju se redom; svaki prijemnik može da prosledi događaj sledećem prijemniku ili da potpuno obustavi njegovu obradu.

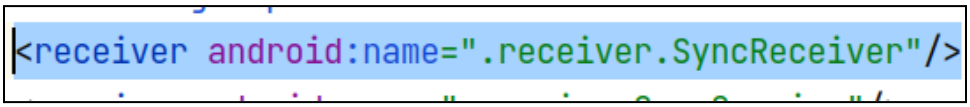
onReceive:

Prijemnik poruka postoji samo u toku izvršavanja metode *onReceive*, te se u njoj ne mogu izvršavati asinhrone operacije (prikazivanje dijaloga, vezivanje za servis..).

Ova metoda se poziva iz glavne niti, te duge operacije treba izvršavati u posebnom servisu, koji startuje posebnu nit.

4.1 Pravljenje prijemnika poruka

Pravimo novu klasu *SyncReceiver* koja nasleđuje *BroadcastReceiver* i u metodi *onReceive* kreiramo ponašanje. Isto kao i servise, prijemnike poruka navodimo u *AndroidManifest* datoteci (slika 16).



Slika 16. Navođenje prijemnika poruka u AndroidManifest-u

Pre nego što pogledamo šta će tačno raditi naš prijemnik poruka, nameće se pitanje kako ćemo uopšte da mu šaljemo poruke?

U našem asinhronom zadatku, koji smo napravili u prethodnom poglavlju, u okviru metode *handler.post()* napravili smo *Intent* i definisali akciju „SYNC DATA“ (slika 17). Akciju smo definisali zato što jedan prijemnik poruka može da prima više poruka iz aplikacije. Uz tu poruku smo definisali i RESULT_CODE parametar koji prosleđujemo intent-u sa statusom konekcije. Na samom kraju, pozivamo metodu *sendBroadcast* i prosleđujemo joj kreiranu nameru (*intent*).

```
handler.post(() -> {
    //UI Thread work here
    Log.i( tag: "REZ",  msg: "UI Thread work here");
    Intent ints = new Intent(HomeActivity.SYNC_DATA);
    int intsStatus = CheckConnectionTools.getConnectivityStatus(getApplicationContext());
    ints.putExtra(RESULT_CODE, intsStatus);
    getApplicationContext().sendBroadcast(ints);
});
```

Slika 17. Metoda asinhronog zadatka koja šalje poruku prijemniku poruka

Ako se vratimo na našu klasu *SyncReceiver,* ona u metodi *onReceive* prima nameru (intent), koja će čuvati akciju i podatke, koji su stigli (slika 18). Na prethodnoj slici smo videli da asinhroni zadatak prosleđuje prijemniku poruka nameru, a sada vidimo i gde ta namera stiže.
Naš prijemnik poruka će reagovati ako je dobio "SYNC DATA", te u if-u proveravamo da li je prijemniku upravo takva poruka stigla.

```
@Override
public void onReceive(Context context, Intent intent) {
    Log.i( tag: "REZ",  msg: "onReceive");
    //...
    NotificationManagerCompat notificationManager = NotificationManagerCompat.from(context);
    // notificationId is a unique int for each notification that you must define
    NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(context, SYNC_CHANNEL_ID);

    /*...*/
    if (intent.getAction().equals(HomeActivity.SYNC_DATA)) {
        int resultCode = intent.getExtras().getInt(SyncService.RESULT_CODE);
        Bitmap bm;
        Intent wiFiintent = new Intent(Settings.ACTION_WIFI_SETTINGS);
        PendingIntent pIntent = PendingIntent.getActivity(context, NOTIFICATION_ID, wiFiintent, PendingIntent.FLAG_IMMUTABLE);
```

Slika 18. Metoda *onReceive*

U slučaju da nam je stigla poruka „SYNC DATA", preuzimamo sadržaj namere. Dalje ponašanje prijemnika poruke zavisi od sadržaja koji je stigao, tj. informacije da li je uređaj povezan na internet i ako jeste na koji način (slika 19). U našem slučaju, korisniku pristiže notifikacija sa informacijom o konekciji.

```
if (intent.getAction().equals(HomeActivity.SYNC_DATA)) {
    int resultCode = intent.getExtras().getInt(SyncService.RESULT_CODE);
    Bitmap bm;
    Intent wiFiintent = new Intent(Settings.ACTION_WIFI_SETTINGS);
    PendingIntent pIntent = PendingIntent.getActivity(context, NOTIFICATION_ID, wiFiintent,  flags: 0);

    if (resultCode == CheckConnectionTools.TYPE_NOT_CONNECTED) {
        bm = BitmapFactory.decodeResource(context.getResources(), R.drawable.ic_action_network_wifi);
        mBuilder.setSmallIcon(R.drawable.ic_action_about);
        mBuilder.setContentTitle("Automatic Sync problem");
        mBuilder.setContentText(context.getString(R.string.no_internet));
        mBuilder.setContentIntent(pIntent);
        mBuilder.addAction(R.drawable.ic_action_network_wifi, context.getString(R.string.turn_wifi_on), pIntent);
        mBuilder.setPriority(NotificationCompat.PRIORITY_DEFAULT);

    } else if (resultCode == CheckConnectionTools.TYPE_MOBILE) {
        bm = BitmapFactory.decodeResource(context.getResources(), R.drawable.ic_action_network_cell);
        mBuilder.setSmallIcon(R.drawable.ic_action_warning);
        mBuilder.setContentTitle(context.getString(R.string.autosync_warning));
        mBuilder.setContentText(context.getString(R.string.connect_to_wifi));
        mBuilder.addAction(R.drawable.ic_action_network_wifi, context.getString(R.string.turn_wifi_on), pIntent);
    } else {
        bm = BitmapFactory.decodeResource(context.getResources(), R.drawable.ic_launcher_foreground);
        mBuilder.setSmallIcon(R.drawable.ic_action_refresh_w);
        mBuilder.setContentTitle(context.getString(R.string.autosync));
        mBuilder.setContentText(context.getString(R.string.good_news_sync));
    }
    mBuilder.setLargeIcon(bm);
```

Slika 19. Ponašanje prijemnika poruke u zavisnosti od sadržaja koji mu je stigao

Bitno je još napomenuti da smo u metodi *onCreate* klase *MainActivity* pozvali napravljenu metodu *setUpReceiver* (slika 20). Ta metoda inicijalizuje prijemnik poruka i definiše nameru, koju želimo da izvršavamo kada dođe za to vreme.

```
139     private void setUpReceiver(){
140         syncReceiver = new SyncReceiver();
141
142         //definisemo manager i kazemo kada je potrebno da se ponavlja
143         /*...*/
152         Intent alarmIntent = new Intent( packageContext: this, SyncService.class);
153         pendingIntent = PendingIntent.getService( context: this,  requestCode: 0, alarmIntent,  flags: 0);
154
155         //...
157         manager = (AlarmManager)getSystemService(Context.ALARM_SERVICE);
158     }
```

Slika 20. Metoda *setUpReceiver*

5. Zakazivanje zadataka

Timer je komponenta koja služi za zakazivanje jednokratnih zadataka ili zadataka koji se ponavljaju. Svaki timer ima jednu nit koja zadatke izvršava sekvencijalno. Kada aplikacija nije aktivna, timer neće izvršiti svoj posao.

Za demonstraciju *timer*-a (slika 21) koristimo uvodni ekran za korisnika *(splash screen*). Timer zakazujemo tako što pozivamo metodu *schedule* i prosleđujemo joj *TimerTask*, sa metodom *run*, i vreme koliko dugo *timer* treba da čeka da bi se posao izvršio. U metodi *run* smo napravili eksplicitnu nameru, prelazak sa aktivnosti *SplashScreenActivity* na aktivnost *MainActivity*.

```
public class SplashScreenActivity extends Activity {
    public Timer timer = new Timer();

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.splash);
        int SPLASH_TIME_OUT = 3000;

        /*...*/
        timer.schedule(() -> {
                startActivity(new Intent( packageContext: SplashScreenActivity.this, MainActivity.class));
                finish(); // da nebi mogao da ode back na splash
        }, SPLASH_TIME_OUT);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        timer.cancel();
    }
}
```

Slika 21. Primer upotrebe *timer*-a

Klasa *AlarmManager* nam omogućuje da pristupimo sistemskom alarmu i da pokrenemo aplikaciju u nekom trenutku u budućnosti.

U metodi *setUpReceiver* koristimo *AlarmManager* i definišemo kada je potrebno da se ponavlja (slika 22). U poslednjoj liniji ove metode dobavljamo instancu sistemskog *AlarmManager*-a.

```
    private void setUpReceiver(){
        syncReceiver = new SyncReceiver();

        //definisemo manager i kazemo kada je potrebno da se ponavlja
        /*...*/
        Intent alarmIntent = new Intent( packageContext: this, SyncService.class);
        pendingIntent = PendingIntent.getService( context: this,  requestCode: 0, alarmIntent,  flags: 0);

        //...
        manager = (AlarmManager)getSystemService(Context.ALARM_SERVICE);
    }
```

Slika 22. Primer upotrebe *AlarmManager* klase

Kod na slici 23 pokazuje primer definisanja manager-a čiji zadaci treba da se periodično ponavljaju.

Svaki put kada dođe trenutak da se pokrene zadatak, emitovaće se objekat klase *Intent* koji će reći OS-u šta treba da se uradi.

Na prvoj liniji na slici 23 definišemo na koliko vremena treba neki zadatak da se uradi, a u narednim linijama kako da manager reaguje. Manager je definisan tako da je u režimu ponavljanja (prvi parametar), da kreće odmah da meri vreme (drugi parametar), da reaguje na svaki minut (treći parametar - *interval*), i kao poslednji parametar smo mu rekli šta treba da uradi kada se alarm isključi.

Svi zadaci koji se ponavljaju nisu uvek egzaktni, što znači da se najverovatnije neće izvršiti tačno na svakih n minuta, već na n + x minuta, gde je x neko malo odstupanje.

```
int interval = CheckConnectionTools.calculateTimeTillNextSync( minutes: 1);

//...
manager.setRepeating(AlarmManager.RTC_WAKEUP, System.currentTimeMillis(), interval, pendingIntent);
Toast.makeText( context: this,  text: "Alarm Set", Toast.LENGTH_SHORT).show();
```

Slika 23. Definisanje zadatka koji će se izvršavati periodično

6. Domaći

Domaći se nalazi na *Canvas-u (canvas.ftn.uns.ac.rs)* na putanji *Вежбе/05 Задатак.pdf*.

Primer možete preuzeti na sledećem linku:
https://gitlab.com/mobilne-aplikacije/mobilne-aplikacije-siit-2024-25

Za dodatna pitanja možete se obratiti asistentima:

- Svetlana Antešević (svetlanaantesevic@uns.ac.rs)
- Jelena Matković (matkovic.jelena@uns.ac.rs)