



MOBILNE APLIKACIJE

Vežbe 5

Servisi i prijemnici poruka

2022/2023

Sadržaj

1. Servisi	3
1.1 Životni ciklus servisa	3
1.2 Pravljenje servisa	4
2. Asinhroni zadaci	6
2.1 Pravljenje asinhronog zadatka	6
2.2 Handler i Looper	7
3. Prijemnici poruka	8
3.1 Pravljenje prijemnika poruka	8
4. Zakazivanje zadataka	11
5. Domaći	13

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.

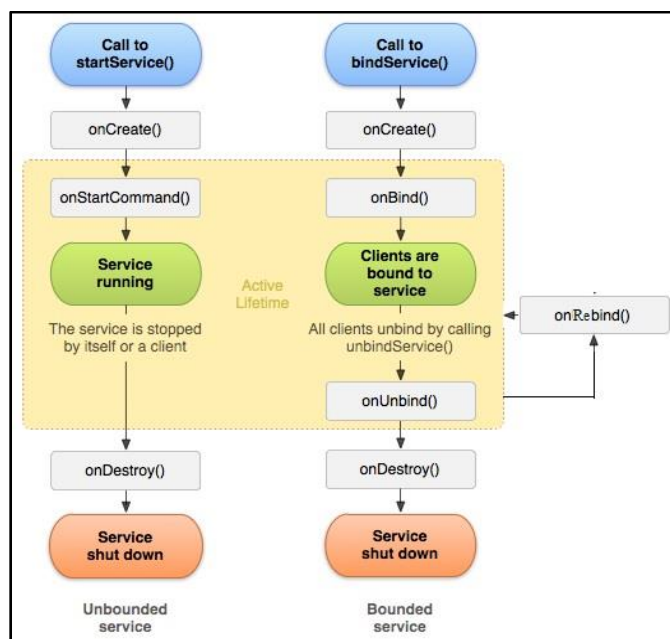
Postoje 2 vrste servisa:

- Servis koji je pokrenut
- Servis koji je vezan

Servis koji je pokrenut se izvršava neodređeno vreme i zaustavlja se kada završi operaciju. **Servis koji je vezan** se izvršava samo dok je neka druga komponenta vezana za njega. 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 završak 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.

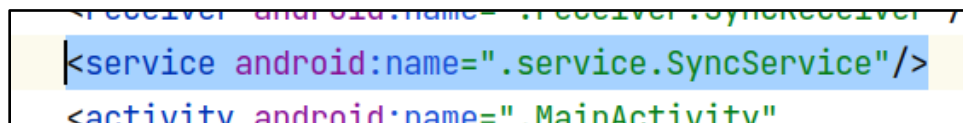
onDestroy

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 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).



```
<service android:name=".service.SyncService"/>
```

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 *ReviewerTools* 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 36).

U slučaju da je uslov zadovoljen, tj. da je uređaj povezan na internet, pokrećemo asinhroni zadatak (linija 43).

Na kraju na liniji 68 pozivamo metodu *stopSelf*, koja će zaustaviti servis.

```

23
24  /*...*/
29  @Override
30  public int onStartCommand(Intent intent, int flags, int startId) {
31      Log.i( tag: "REZ", msg: "SyncService onStartCommand");
32      /*...*/
36      int status = ReviewerTools.getConnectivityStatus(getApplicationContext());
37      /*...*/
41      if(status == ReviewerTools.TYPE_WIFI || status == ReviewerTools.TYPE_MOBILE){
42          // Alternativa za SyncTask
43          executor.execute() -> {
44              //Background work here
45              Log.i( tag: "REZ", msg: "Background work here");
46              try {
47                  Thread.sleep( millis: 1000);
48              } catch (InterruptedException e) {
49                  e.printStackTrace();
50              }
51              handler.post() -> {
52                  //UI Thread work here
53                  Log.i( tag: "REZ", msg: "UI Thread work here");
54                  Intent ints = new Intent(MainActivity.SYNC_DATA);
55                  int intsStatus = ReviewerTools.getConnectivityStatus(getApplicationContext());
56                  ints.putExtra(RESULT_CODE, intsStatus);
57                  getApplicationContext().sendBroadcast(ints);
58              });
59          });
60      /*...*/
62  }
63      /*...*/
68      stopSelf();
69
70      /*...*/
74      return START_NOT_STICKY;
75  }

```

Slika 3. Metoda servisa *onStartCommand*

2. 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.

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 (oni se serijalizuju).

2.1 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*.

Primer starog načina kreiranja asinhronog zadatka dat je u foldery *sync*, klasa *SyncTask*.

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 4.

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

Slika 4. Metoda *newSingleThreadExecutor()*

Pomoću instance *executor* moguće je izvršiti asinhron zadatak pomoću metode *execute()* u okviru koje je moguće implementirati *Runnable* 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 5).

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

Slika 5. 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 6).

```
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 6. Metoda `runOnUiThread()`

2.2 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*) unutar niti `run()` metode. *Looper* održava svoju nit živom. 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. Možemo kreirati *Handler* koji je vezan za određeni *Looper* na sledeći način dat na slici 7. Metoda `post()` kreira poruku i dodaje je na kraj reda *Looper*-a.

```
ExecutorService executorService = Executors.newSingleThreadExecutor();  
Handler handler = new Handler(Looper.getMainLooper());  
  
executorService.execute(new Runnable() {  
    @Override  
    public void run() {  
        Log.i( tag: "REZ", msg: "Background work here");  
        handler.post(new Runnable() {  
            @Override  
            public void run() {  
                Log.i( tag: "REZ", msg: "UI Thread work here");  
            }  
        });  
    }  
});
```

Slika 7. Kreiranje *handler*-a i metoda `post()`

3. 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.

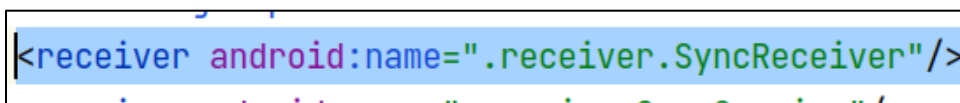
onReceive

Prijemnik poruka postoji samo u toku izvršavanja metode *onReceive*, te se u njoj ne mogu izvršavati asinhronne 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.

3.1 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 8).



```
<receiver android:name=".receiver.SyncReceiver"/>
```

Slika 8. 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 9). Akciju smo definisali zato što jedan prijemnik poruka može da prima više poruka iz aplikacije. Uz tu poruku smo definisali i *RESULT_CODE*. 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(MainActivity.SYNC_DATA);
    int intsStatus = ReviewerTools.getConnectivityStatus(getApplicationContext());
    ints.putExtra(RESULT_CODE, intsStatus);
    getApplicationContext().sendBroadcast(ints);
};

```

Slika 9. 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 10). Na prethodnoj slici smo videli da asinhroni zadatak prosleđuje prijemniku 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.

```

37  /*...*/
41  @Override
42  public void onReceive(Context context, Intent intent) {
43      Log.i( tag: "REZ", msg: "onReceive");
44      //...
46      NotificationManagerCompat notificationManager = NotificationManagerCompat.from(context);
47      // notificationId is a unique int for each notification that you must define
48      NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(context, CHANNEL_ID);
49      /*...*/
58      if(intent.getAction().equals(MainActivity.SYNC_DATA)){
59          int resultCode = intent.getExtras().getInt(SyncService.RESULT_CODE);
60          Bitmap bm;
61

```

Slika 10. 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 11).

```

58     if(intent.getAction().equals(MainActivity.SYNC_DATA)){
59         int resultCode = intent.getExtras().getInt(SyncService.RESULT_CODE);
60         Bitmap bm;
61         Intent wiFiIntent = new Intent(Settings.ACTION_WIFI_SETTINGS);
62         PendingIntent pIntent = PendingIntent.getActivity(context, requestCode: 0, wiFiIntent, flags: 0);
63
64         if(resultCode == ReviewerTools.TYPE_NOT_CONNECTED){
65             bm = BitmapFactory.decodeResource(context.getResources(), R.drawable.ic_action_network_wifi);
66             mBuilder.setSmallIcon(R.drawable.ic_action_error);
67             mBuilder.setContentTitle("Automatic Sync problem");
68             mBuilder.setContentText("Bad news, no internet connection");
69             mBuilder.setContentIntent(pIntent);
70             mBuilder.addAction(R.drawable.ic_action_network_wifi, "Turn wifi on", pIntent);
71             mBuilder.setPriority(NotificationCompat.PRIORITY_DEFAULT);
72
73         }else if(resultCode == ReviewerTools.TYPE_MOBILE){
74             bm = BitmapFactory.decodeResource(context.getResources(), R.drawable.ic_action_network_cell);
75             mBuilder.setSmallIcon(R.drawable.ic_action_warning);
76             mBuilder.setContentTitle("Automatic Sync warning");
77             mBuilder.setContentText("Please connect to wifi to sync data");
78             mBuilder.addAction(R.drawable.ic_action_network_wifi, "Turn wifi on", pIntent);
79         }else{
80             bm = BitmapFactory.decodeResource(context.getResources(), R.drawable.ic_launcher);
81             mBuilder.setSmallIcon(R.drawable.ic_action_refresh_w);
82             mBuilder.setContentTitle("Automatic Sync");
83             mBuilder.setContentText("Good news, everything is sync now.");
84         }
85         mBuilder.setLargeIcon(bm);
86         // notificationID allows you to update the notification later on.
87         notificationManager.notify(NOTIFICATION_ID, mBuilder.build());
88     }

```

Slika 11. 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 12). 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         //definise manager i kazemo kada je potrebno da se ponavlja
143         /*...*/
144
145         Intent alarmIntent = new Intent( packageContext: this, SyncService.class);
146         pendingIntent = PendingIntent.getService( context: this, requestCode: 0, alarmIntent, flags: 0);
147
148         /*...*/
149
150         manager = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
151     }

```

Slika 12. Metoda *setUpReceiver*

4. 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 13) 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*.

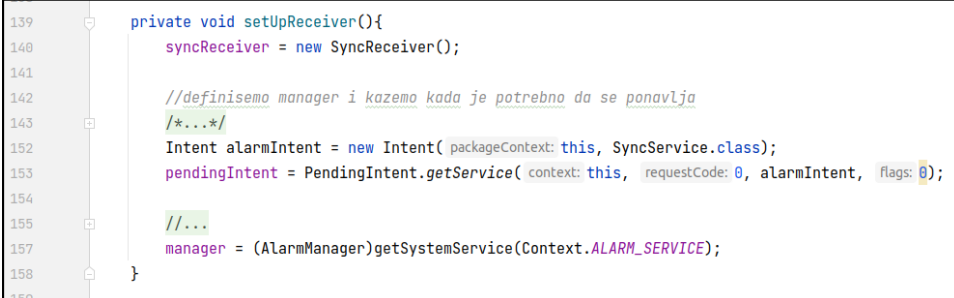


```
3 public class SplashScreenActivity extends Activity {
4     public Timer timer = new Timer();
5
6     @Override
7     protected void onCreate(Bundle savedInstanceState)
8     {
9         super.onCreate(savedInstanceState);
10        setContentView(R.layout.splash);
11        int SPLASH_TIME_OUT = 3000;
12
13        /*...*/
14        timer.schedule(() -> {
15            startActivity(new Intent( packageContext: SplashScreenActivity.this, MainActivity.class));
16            finish(); // da nebi mogao da ode back na splash
17        }, SPLASH_TIME_OUT);
18    }
19
20    @Override
21    protected void onDestroy() {
22        super.onDestroy();
23        timer.cancel();
24    }
25 }
```

Slika 13. 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 14). U poslednjoj liniji ove metode dobavljamo instancu sistemskog *AlarmManager*-a



```
139 private void setUpReceiver(){
140     syncReceiver = new SyncReceiver();
141
142     //definise manager i kazemo kada je potrebno da se ponavlja
143     /*...*/
144     Intent alarmIntent = new Intent( packageContext: this, SyncService.class);
145     PendingIntent pendingIntent = PendingIntent.getService( context: this, requestCode: 0, alarmIntent, flags: 0);
146
147     /*...*/
148     manager = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
149 }
```

Slika 14. Primer upotrebe *AlarmManager* klase

Kod na slici 15 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 15 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* koji je definisan na prvoj liniji), 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.

```
176  /*...*/
184  int interval = ReviewerTools.calculateTimeTillNextSync( minutes: 1);
185
186  //...
191  manager.setRepeating(AlarmManager.RTC_WAKEUP, System.currentTimeMillis(), interval, pendingIntent);
192  Toast.makeText( context: this, text: "Alarm Set", Toast.LENGTH_SHORT).show();
```

Slika 15. Definisanje zadatka koji će se izvršavati periodično

5. Domaći

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

Primer *Vezbe5* i *Vezbe5_Primer2* možete preuzeti na sledećem linku:

<https://gitlab.com/antesevicceca/mobilne-aplikacije>

Za dodatna pitanja možete se obratiti asistentima:

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