Laboratoire 5

Services, notifications et récepteurs d'événements

Objectifs d'apprentissage

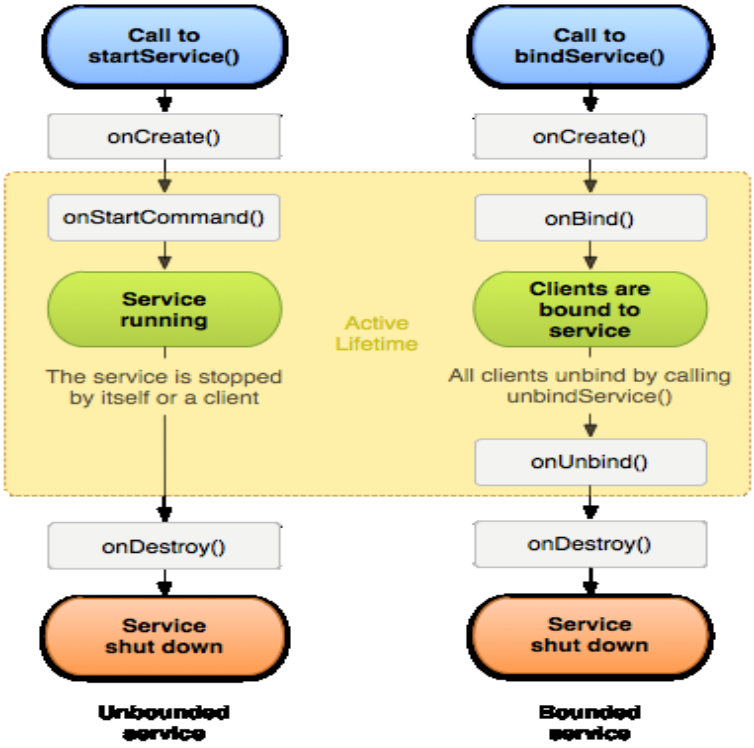
- Comprendre le cycle de vie d'un service
- Créer un service
- Lancer, arrêter un service directement
- Se connecter à un service
- Se déconnecter d'un service
- Créer et envoyer une notification
- Créer et utiliser un récepteur d'évènements

1. Cycle de vie d'un service

Un service est un composant applicatif qui ne possède pas d'interface graphique et qui s'exécute en arrière-plan. Son utilisation peut être limitée à l'application où il est défini ou proposée à d'autres applications. Ce type d'information est précisé dans le fichier *manisfest* lors de la déclaration du service. Un service peut être lancé de deux façons :

- Directement : en appelant la méthode ***startService()*** qui prend en argument un objet *intent* implicite ou explicite, permettant d'identifier le service à démarrer. Le service peut être arrêté à tout moment en appelant la méthode ***stopService().*** Il peut aussi mettre fin à sa propre exécution à l'aide la méthode ***stopSelf().***
- En établissant une connexion ***(bindService())*** *:* Ceci permet une communication client-serveur entre les composants. Dans ce cas, le service est détruit quand tous les clients sont déconnectés.

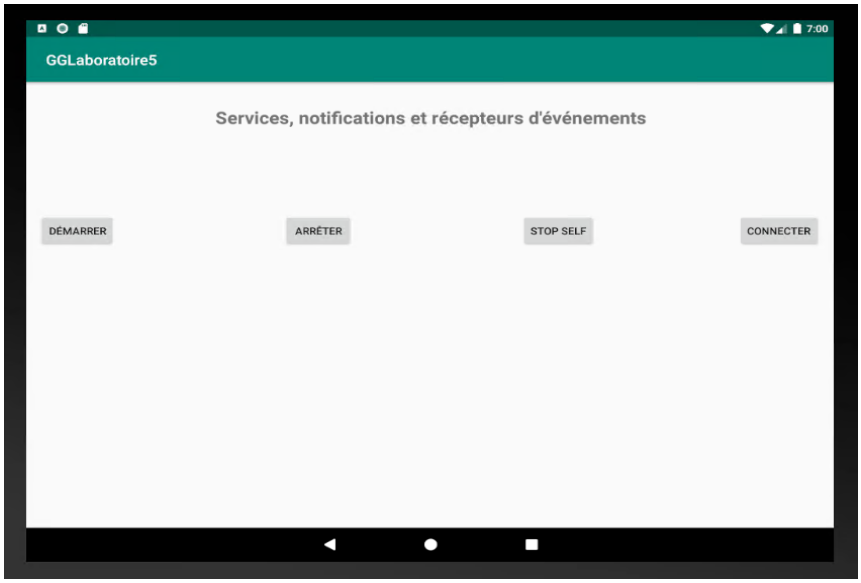
  Les deux méthodes peuvent être utilisées en même temps.



Important : Un service s'exécute par défaut dans le *thread* principal de l'application. Il est fortement conseillé de créer d'autres *threads* pour vos services afin de laisser le *thread* principal pour les interactions avec l'utilisateur.

1.1 Lancement et arrêt d'un service

- Créer un nouveau projet **Laboratoire 5**. Modifiez la vue ***activity_main*** pour avoir une interface semblable à ce qui suit :



- Programmer le clic des boutons en affichant simplement avec la fonction Log.i le texte ***MainActivity***, ***nom du bouton cliqué*** où *MainActivity* constitue le filtre (TAG) pour l'affichage. Affichez également un toast avec les mêmes informations. Tous vos boutons doivent être associés à la même fonction clickHandler suivante.

```
fun clickHandler(view: View) {
    Log.i(TAG, (view as Button).text.toString())
    Toast.makeText(this, "dans ${object
{}.javaClass.enclosingMethod.name}", Toast.LENGTH_SHORT)
        .show()
}
```

- Exécuter votre application. Vérifier que tous vos boutons répondent au clic.
- Ajouter une classe ***TestService***, sous classe de *Service* à votre projet. Programmer les fonctions de rappel du service comme suit:

```
import android.app.Service
import android.content.Intent
import android.os.IBinder
import android.util.Log
import android.widget.Toast

class TestService : Service() {
    val TAG: String = TestService::class.java.canonicalName

    override fun onBind(intent: Intent): IBinder {
        throw UnsupportedOperationException("Not yet implemented")
    }

    override fun onCreate() {
        super.onCreate()
        Log.i(TAG, "dans ${object {}.javaClass.enclosingMethod.name}")
        Toast.makeText(this, "dans ${object {}.javaClass.enclosingMethod.name}",
Toast.LENGTH_SHORT).show()
    }

    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
        Toast.makeText(this, "dans ${object {}.javaClass.enclosingMethod.name}:
startId=$startId", Toast.LENGTH_SHORT).show()
        return super.onStartCommand(intent, flags, startId)
    }
}
```

- Ajouter le code suivant à la fin de la fonction clickHandler pour qu'au clic du bouton ***Démarrer*** le service ***TestService*** soit lancé et qu'au clic du bouton ***Arrêter*** le service soit arrêté.

```
val intent = Intent(this, TestService::class.java)
when(view.id){
    R.id.btnStart ->{
        startService(intent)
    }
    R.id.btnStop -> {
        stopService(intent)
    }
}
```

- Déclarer le service dans le fichier AndroidManifest.xml si vous n'avez pas utilisé le template d'Android Studio et créé manuellement la classe TestService.

```
<service
    android:name=".TestService"
    android:enabled="true"
    android:exported="true"></service>
```

- Exécuter. Observer l'exécution des méthode ***onCreate()*** et ***onStartCommand()*** du service. Noter la valeur du paramètre *startId*.
- Appuyez plusieurs fois sur le bouton ***Démarrer***. Observer les nouvelles valeurs du paramètre ***startId***.
- Quitter l'application. Y revenir, puis cliquer à nouveau sur le bouton ***Démarrer***. Remarquer que le service n'a pas été arrêté. Il ne **repasse pas** par ***onCreate()***. Il continue à incrémenter la valeur du paramètre ***startId*** de la méthode ***onStartCommand()***
- Compléter la classe ***TestService*** avec les méthodes suivantes :

```
override fun onDestroy() {
    Toast.makeText(this, "dans ${object
{}.javaClass.enclosingMethod.name}", Toast.LENGTH_SHORT).show()
    super.onDestroy()
}
```

- Exécuter à nouveau. Appuyez sur le bouton ***Démarrer*** (une ou plusieurs fois). Cliquer sur le bouton ***Arrêter***. Obverser l'exécution de la fonction de rappel ***onDestroy()*** du service. Redémarrer le service. Observer le comportement.
- Ajoutez la variable de classe suivante à la classe TestService:

```
class TestService : Service() {
    val TAG: String = TestService::class.java.canonicalName
```

- Pour programmer le bouton ***stopSelf*** qui permet au service de mettre fin à sa propre exécution, ajouter le code suivant à la fonction ***onStartCommand()*** de la classe ***TestService*** après la ligne affichant le toast.

```
val stop = intent?.getBooleanExtra(CLE_STOP, false)
if (stop!!) {
    Toast.makeText(this, "stopSelf", Toast.LENGTH_SHORT) .show()
    stopSelf()
}
```

- Ajouter le code suivant à la fonction clickHandler pour gérer le clic du bouton ***stopSelf*** de la classe ***MainActivity***

```
intent.putExtra(TestService.CLE_STOP, true)
        startService(intent)
```

- Exécuter. Observer le comportement.

1.2 Connexion et déconnexion d'un service

- Modifiez la fonction ***clickHandler*** de la classe ***MainActivity*** pour gérer le clic du bouton ***Connecter*** comme suit:

```
R.id.btnConnect -> {
    if (mService == null) {
        bindService(intent, this.mConnexion, Context.BIND_AUTO_CREATE)
    } else {
        unbindService(mConnexion)
        mService = null
        (view as Button).setText(R.string.btnConnectTxt)
    }
}
```

- Compléter la classe ***TestService*** avec la déclaration des objets ***mService*** et ***mConnection*** comme suit:

```
private var mService: TestService? = null

private val mConnexion = object : ServiceConnection {
    override fun onServiceConnected(arg0: ComponentName, arg1: IBinder){
        Toast.makeText(this@MainActivity, "dans ${object
{}.javaClass.enclosingMethod.name}", Toast.LENGTH_SHORT).show()
        mService = (arg1 as TestService.MonServiceBinder).getService()
        btnConnect.setText(R.string.btnDisconnectTxt)
    }

    override fun onServiceDisconnected(arg0: ComponentName) {
        Toast.makeText(this@MainActivity, "dans ${object
{}.javaClass.enclosingMethod.name}", Toast.LENGTH_SHORT).show()
        mService = null
        btnConnect.setText(R.string.btnConnectTxt)
    }
}
```

IMPORTANT : Ne pas oublier de déclarer les chaines de caractères dans le fichier ressource strings.

- Modifiez la classe ***TestService*** comme suit :

1. Ajoutez la classe interne ***MonServiceBinder*** comme suit à la classe ***TestService***. Cette classe est une spécialisation de la classe ***Binder***. Elle retourne une instance de la classe ***TestService*** en implémentant la méthode ***getService()***.

```
inner class MonServiceBinder: Binder() {
    fun getService(): TestService {
        return this@TestService
    }
}
```

2. Ajoutez la variable ***mBinder*** à la classe TestService comme suit:

```
//créer une référence vers l'instance du service TestService
private val mBinder : IBinder = MonServiceBinder()
```

3. Modifiez la fonction onBind() de la classe TestService comme suit:

```
override fun onBind(intent: Intent): IBinder {
    Toast.makeText(this, "dans ${object
{}.javaClass.enclosingMethod.name}", Toast.LENGTH_SHORT).show()
    return mBinder
}
```

4. Ajoutez la function onUnbind à la classe TestService comme suit:

```
override fun onUnbind(intent: Intent): Boolean {
    Toast.makeText(this, "dans ${object
{}.javaClass.enclosingMethod.name}", Toast.LENGTH_SHORT).show()
    return true
}
```

- Exécuter à nouveau. Cliquer sur le bouton ***Connecter***. Observer la passage du service par les méthodes ***onCreate()***, ***onBind()*** et ***onServiceConnected()***
- Cliquer sur *Démarrer.* Noter qu'il ne repasse plus pas ***onCreate()***. Il exécute *directement* ***onStartCommand()***.
- Essayer d'arrêter le service une fois connecté…
- Appuyez sur le bouton ***Déconnecter***. Observer le comportement.
- Essayer plusieurs séquences de clics pour bien comprendre le cycle de vie d'un service.

2. Notifications

La barre de notification ou barre de statut permet aux applications tournant en tâche de fond d'avertir l'utilisateur sans perturber l'utilisation courante de l'appareil. Cette barre reçoit et stocke les notifications qui lui sont envoyées par les applications.

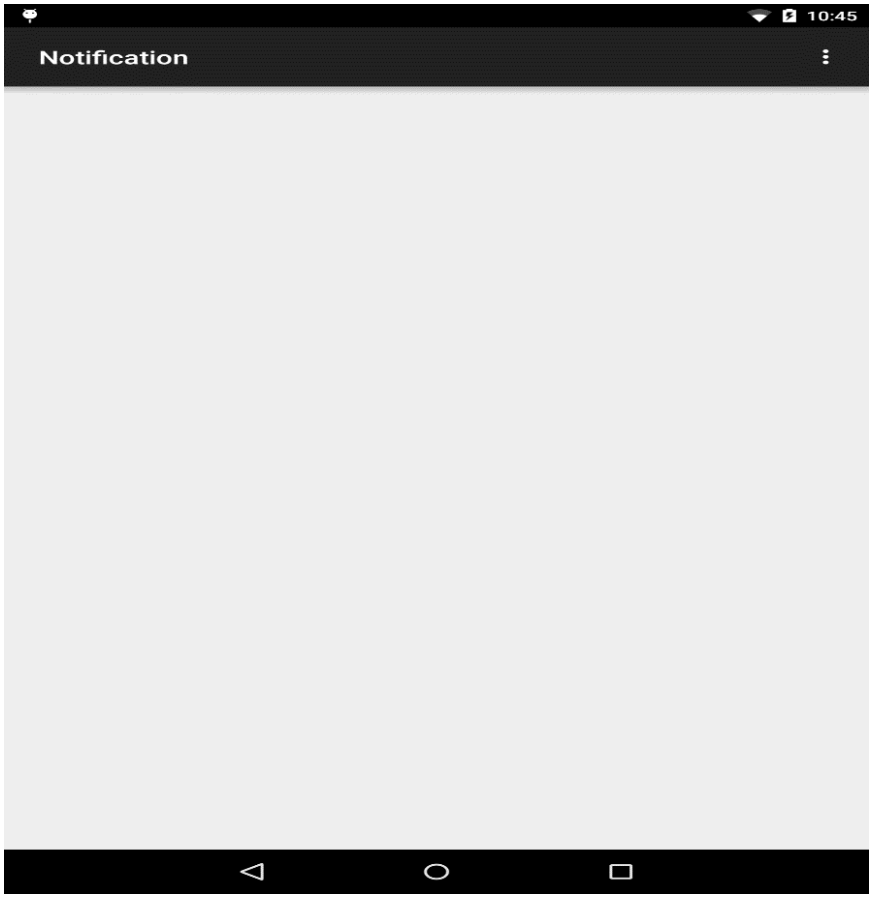
2.1 Création et envoi d'une notification

- Compléter l'interface ***MainActivity*** pour avoir ce qui suit :

- Ajouter le code suivant au clic du bouton *ajouter une notification.* Bien vous documenter sur chacune des opérations pour bien comprendre le processus de création et d'envoi d'une notification. Ajouter toutes les chaînes nécessaires dans le fichier ressource strings.xml.
- **Le titre de la notification étant : test notification** et le texte **: Un clic ouvre l'activité *Notification.*** Utilisez un logo étoilé comme icône à gauche de la notification. Modifiez la fonction **clickHandler** comme suit pour gérer le clic du bouton ***Ajouter une notification***.

```
R.id.btnAjouterNotification->{
    val mLogo = BitmapFactory.decodeResource(resources,
R.drawable.ic_baseline_redstar_24)

    val patternVibrations = longArrayOf(0, 50, 50, 50, 50, 50, 50, 50, 50, 50,
50, 50, 50, 50)
    // Création de la notification. Noter cette nouvelle façon de créer un objet
    val notificationBuilder = Notification.Builder(this)
        .setLargeIcon(mLogo)
        .setAutoCancel(true) // Notification disparaît quand l'utilisateur la
touche.
        .setSmallIcon(R.drawable.ic_launcher_foreground)
        .setContentTitle(getString(R.string.notification_titre))
        .setContentText(getString(R.string.notification_text))
        .setNumber(1)
        .setVibrate(patternVibrations)

    //intent pour la notification
    val notificationIntent: Intent = Intent(this,
NotificationActivity::class.java);
    notificationIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    //Noter l'utilisation d'un PendingIntent (Intention en attente).
    val pendingIntent: PendingIntent = PendingIntent.getActivity(this, 0,
notificationIntent, 0);
    notificationBuilder.setContentIntent(pendingIntent);


    val notification: Notification? = notificationBuilder.notification
    Toast.makeText(
        this, when (notification) {
            null -> R.string.notification_erreur
            else -> R.string.notification_ajoutee
        }, Toast.LENGTH_LONG
    ).show()

    if (notification != null) {
        val notificationManager: NotificationManager =
            getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
        notificationManager.notify(0, notification)
    }
}
```

- Ajouter l'activité NotificationActivity à votre projet (new/Activity..) . Cette activité sera appelée quand on clique sur la notification. Pour la distinguer, il suffit de lui donner le titre *Notification* (propriété label). Sa vue ne contient aucun composant.



- Exécuter. **Une erreur apparaît sur les versions 4.0.X et moins**.
- Pour corriger la situation, il faut ajouter dans le fichier ***AndroidManifest*** la permission pour les vibrations :

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

Cette balise est dans la balise *manifest* mais en dehors de la balise *application*

- Exécuter. Cliquer sur le bouton ***Ajouter une notification***. Vérifier que l'activité ***NotificationActivity*** s'affiche. Noter que la notification disparaît de la barre de statut juste après le clic.

2.2 Utilisation d'une notification dans le cas d'un service

Nous allons compléter la classe *TestService* pour permettre l'affichage d'une notification informant que le service est bien démarré. Contrairement à la notification créée en 2.1, celle-ci sera persistante. Elle ne pourra donc pas être supprimée par l'utilisateur. Il faudra s'assurer que l'application la supprime quand le service sera détruit.

- Ajouter la méthode suivante à la classe *TestService*

```
private fun afficherNotification(nb: Int) {
    val mLogo = BitmapFactory.decodeResource(resources,
R.drawable.ic_baseline_redstar_24)
    val patternVibrations = longArrayOf(0, 50, 50, 50, 50, 50, 50, 50, 50, 50,
50, 50, 50, 50)
    val notificationBuilder = Notification.Builder(this)
        .setLargeIcon(mLogo)
        .setOngoing(true) //persistante. ne peut pas être supprimée par
l'utilisateur
        .setSmallIcon(R.drawable.ic_launcher_foreground)
        .setContentTitle(getString(R.string.notification_service_titre))
        .setContentText(getString(R.string.notification_service_text))
        .setNumber(1)
        .setVibrate(patternVibrations)

    val notification = notificationBuilder.notification
    if (notification != null) {
        val notificationManager = getSystemService(Context.NOTIFICATION_SERVICE)
as NotificationManager
        notificationManager.notify(0, notification)
    }
}
```

- Dans le fichier strings.xml, ajouter les chaînes *notification_service_titre* et *notification_service_text* avec comme valeurs respectives *Le service TestService est démarré* et *cette notification ne peut être supprimée par l'utilisateur*
- Ajouter l'instruction suivante dans la méthode *onStartCommand(Intent intent, int flags, int startId)*
  afficherNotification(startId);

- Pour supprimer la notification de la barre de statut quand le service est arrêté, ajouter le code suivant dans la méthode

```
override fun onDestroy() {
    Toast.makeText(this, "dans ${object
{}.javaClass.enclosingMethod.name}", Toast.LENGTH_SHORT).show()
    val notificationManager: NotificationManager =
        getSystemService(Context.NOTIFICATION_SERVICE) as
NotificationManager
    notificationManager.cancel(0)
    super.onDestroy()
}
```

- Afin de permettre aussi la notification lorsque le service est démarré via une connexion (binding), ajouter l'instruction suivante dans la méthode *onBind(Intent intent)*

  afficherNotification(0);

- Tester votre application en démarrant le service, puis l'arrêter pour faire disparaître la notification de la barre de statut. Refaire l'opération à l'aider d'une connexion au service.
- Démarrer à nouveau le service. Quitter l'application. Relancer votre application. Noter la présence de la notification (le service continue à fonctionner en tâche de fond).

3. Récepteurs d'évènements

Un récepteur d'évènements est un composant applicatif dont le rôle consiste uniquement à réagir à des évènements auxquels il est abonné (modèle de conception Observer). Exple: arrivée d'un SMS, batterie faible. Comme le service, il ne possède pas d'interface graphique. Lorsqu'il reçoit un évènement, s'il désire informer l'utilisateur, il doit le faire en utilisant la barre de notification ou en lançant une activité. L'exécution d'un récepteur d'évènements s'opère dans le thread principal du processus de l'application où il est défini. Il ne devrait pas bloquer le thread principal plus de dix secondes.

Les évènements sont produits soit par le système, soit par des applications. Ils sont envoyés aux récepteurs d'évènements susceptibles de recevoir l'évènement donné.

Les évènements sont des objets de type *Intent* décrivant une action qui vient d'être réalisée ou un évènement qui vient de se produire. L'envoi de *l'intent* se fait de manière asynchrone pour ne pas bloquer le composant émetteur de l'évènement.

- Ajouter la classe suivante à vote application

```
class RecepteurEvenements : BroadcastReceiver() {

    override fun onReceive(context: Context, intent: Intent) {
        Toast.makeText(context, R.string.reception_evenement,
            Toast.LENGTH_SHORT).show()
    }
}
```

- Définir la chaîne *reception_evenement* avec la valeur : *onReceive : évènement reçu*
- Programmer le clic du bouton *envoyer un évèment* comme suit :

```
R.id.btnSendBroadcast->{
    val intent = Intent(this, RecepteurEvenements::class.java)
    sendBroadcast(intent)
}
```

- Déclarer le récepteur d'évènements dans le fichier *AndroidManifest* (dans la balise *Application*)

```
<receiver android:name=".RecepteurEvenements">
<intent-filter>
 <action android:name=" votreEspaceDeNom.laboratoire4.EVENEMENT_1" />
</intent-filter>
 </receiver>
```

- Tester votre application.

  Remarque

  Avant de recevoir son premier évènement, le composant est inactif. Il devient actif dès qu'il reçoit un évènement en paramètre de sa méthode *onReceive()* Dès la sortie de cette méthode, le composant redevient inactif. Attention au cas où le récepteur d'évènement lance un thread. Le système pourrait tuer le processus de l'application, donc du thread. Pour éviter ce problème, il est conseillé de lancer le thread depuis un service. Le système détectera le service comme actif et ne tuera donc pas le processus, sauf dans des cas extrêmes de besoin de ressources.