Android studio add service to manifest

I'm not robot

reCAPTCHA

Continue

From RAD Studio Go Up to File File, The New Delphi Projects is creating a platform to develop Android services. The new Android Service Master presents the following options: Local Element Description Service Select this option to create a local service. This is the most typical situation where the Android app interacts directly with the Android service, both work on the same process. This option includes a TAndroidService data module from System.Android.Service, with the necessary events. The choice of this option includes the following line in the Android Manifest file of the Android app associated with the Android service. For more information on local services, see Intentional Local Services select this option to create a local service using intent. service_name Use intentions to handle asynchronous on-demand queries. This option includes a TAndroidIntentService data module from System.Android.Service, with the necessary events. The choice of this option includes the following line in the Android Manifest file of the Android app associated with the Android service. For more information about the services of intent, see Remote Service Select this option to create a remote service. service_name Remote service can be used by other applications other than those where the service is located. This option includes a TAndroidService data module from System.Android.Service, with the necessary events. The choice of this option includes the following line in the Android Manifest file of the Android app associated with the Android service. For more information on remote services, see Intentional Remote Service select this option to create a remote service using intent. service_name Use intentions to handle asynchronous on-demand queries. This option includes a TAndroidIntentService data module from System.Android.Service, with the necessary events. The choice of this option includes the following line in the Android Manifest file of the Android app associated with the Android service. Once you've service_name chosen the option from the master and hit OK, the form designer displays the Delphi project file. The project includes a data module to identify visual components and identify the methods you want the Android service to share with the app For more information on how to link the app to the service, see Uses Uses The file includes the following units: System.SysUtils, System.Classes, System.Android.Service, AndroidApi.JNI.GraphicsContentViewText, Androidapi.JNI.Os; The project_name.dpr file includes: System.Android.ServiceApplication; &lt;Unit_name&gt;; Cm. The service is also a component of an application that can perform long operations in the background. It doesn't provide a user interface. Once launched, the service may continue to run for some time, even after the user switches to another app. In addition, the component can communicate with the service to interact with it and even perform interprocess communication (IPC). For example, a service can process network transactions, play music, perform an I/O file, or interact with a content provider, all from the background. Warning: The service works in the main stream of the hosting process; the service does not create its own thread and does not work in a separate process unless you specify otherwise. You should run any blocking operations on a separate thread in the service to avoid application errors without responding (ANR). Service Types These are three different types of services: the foreground service performs an operation that is visible to the user. For example, an audio app will use a foreground service to play a audio track. Foreground services should display the Notification. Foreground services continue to work even if the user does not interact with the app. When using a foreground service, you need to display a notification so that users are actively aware that the service is working. This notification cannot be rejected if the service is not stopped or removed from the foreground. Learn more about how to set up foreground services in your app. Note: The WorkManager API offers a flexible way of scheduling tasks and can perform these tasks as foreground services if necessary. In many cases, using WorkManager is preferable to using foreground services directly. The help service performs an operation that is not directly noticed by the user. For example, if an application used a service to seal storage, it is usually a background service. Note: If your app targets API 26 or higher, the system imposes restrictions on background services when the app itself is not in the foreground. In most situations, for example, you should not access location information from the background. Instead, plan tasks with WorkManager. A related service is linked when an application component contacts it, causing bindService. The related service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so Interprocess Communication (IPC). The related service only works as long as another component of the application is attached to it. Several components can communicate with the service at the same time, Unit_name./project_name./project_name./ when they are all unrelated, the service collapses. Although this documentation typically discusses started and related services separately, the service can work both ways - it can be launched (to run indefinitely) and also allows binding. It's just a question of whether you're implementing multiple callback methods: onStartCommand to allow components to run it and onBind to allow binding. Whether your service is running, whether it's connected, or both, any component of an application can use the service (even from a separate application) just as any component can use the action to start with intent. However, you can declare the service private in the manifest file and block access from other apps. This is more discussed in the service announcement section in the manifesto. Choosing between service and Flow A service is just a component that can run in the background even if the user doesn't interact with your app, so you should only create the service if that's what you need. If you have to do the work outside the main thread, but only when the user is interacting with your app, you should instead create a new thread in the context of another component of the app. For example, if you want to play music, but only during the time time while your activity is running, you can create a thread in onCreate, start running it onStart, and stop it in onStop. Also consider using thread pools and performers from the java.util.concurrent or kotlin coroutines instead of the traditional Thread class. For more information on how to move execution to background threads, visit Topics on Android. Keep in mind that if you're using the service, it still works in the main stream of the default app, so you still have to create a new thread in the service if it performs intense or blocking operations. The basics for creating a service need to create a subclass of the service or use one of its existing subclasses. Implementation requires overriding some callback methods that handle key aspects of the service lifecycle and providing a mechanism to link components to the service if necessary. These are the most important callback methods that should be override: onStartCommand () The system calls this method, causing startService when another component (such as an action) requests a service launch. When this method runs, the service starts and can run in the background indefinitely. If you implement this, it is your responsibility to stop the service when its work is completed by calling stopSelf or stopService. If you only want to provide a binding, you don't implement this method. onBind () The system calls this method by causing bindService when another component wants to link to the service (for example, to perform RPC). In yours this method needs to provide the interface that customers use to communicate with the service by returning IBinder. You should always implement this method; However, if you don't want to be mandatory, you should go back zero. OnCreate () The system calls this method to perform a one-time set-up of procedures when the service is originally created (before it calls either onStartCommand () or onBind () If the service is already running, this method is not called. OnDestroy () The system causes this method when the service is no longer in use and is destroyed. The service must implement this to clean up any resources, such as threads, registered listeners, or recipients. This is the last call the service receives. If the component starts the service by calling startService (which leads to an onStartCommand call), the service continues to work until it stops with stopSelf or another component stops it by calling stopService. If the component calls bindService to create the service and onStartCommand is not called, the service only works as long as the component is tied to it. Once the service is not connected to all its customers, the system destroys it. The Android system stops the service only when the memory is low and it needs to restore system resources for activities that have the user's focus. If a service is associated with an action that focuses on users, it is less likely to be killed; if the service is declared running in the foreground, it rarely dies. If the service has been up and running for a long time, the system eventually lowers its position on the background task list, and the service becomes very susceptible to killing - if the service is up and running, you have to design it to gracefully handle the reboot system. If the system kills your service, it restarts it as soon as resources become available, but it also depends on the value you return from onStartCommand. For more information on when the system can destroy the service, see the following sections on how you can create startService and bindService methods, as well as how to use them from other components of the application. By announcing the service in the manifesto, you must declare all services in the application's manifesto file, as well as for actions and other components. To announce your service, add an item as a child to the element. Here's an example: &lt;application ...=&gt; &lt;service android:name=. ExampleService&gt;&lt;/service&gt; ... Additional information about the announcement of the service can be obtained in the manifest. There are other attributes that can be included in the item to identify properties, such as permissions, to run the service, and the process in which the service must work. This year, the service is a service. attribute is the only attribute required - it defines the name of the service class. Once the app is published, leave that name unchanged to avoid the risk of code hacking due to a reliance on explicit intentions to start or link the service (read the blog, things that can't change). Warning: To make sure your app is safe, always use explicit intent when starting the Service and do not announce the filters of intent for your services. Using the implicit intention to run a service is a security risk because you can't be sure of a service that responds to intent and the user can't see which service is running. Starting with Android 5.0 (API level 21), the system throws an exception if you call bindService with implicit intent. You can guarantee that your service is only available to your app by turning on the android:exported attribute and setting it to be false. This actually stops other apps from running your service, even when you use explicit intent. Note: Users can see which services are working on their device. If they see a service they don't recognize or trust, they can stop the service. In order to prevent your service being accidentally stopped by users, you need to add an android:description attribute to the element in the app's manifest. In the description, provide a short sentence explaining what the service does and what benefits it provides. Creating a running A service starts that another component starts with a startService call, which results in the onStartCommand service calling. When the service is up and running, it has a life cycle that does not depend on the component that started it. The service can run in the background indefinitely, even if the component that started it is destroyed. Thus, the service must stop itself when its work is completed by calling stopSelf, or another component can stop it by calling stopService. An application component, such as an action, can start the service by calling startService and passing the intent that defines the service and includes any data to use the service. The service receives this intention in the onStartCommand () method. For example, let's say that an action should store some data in an online database. The action can start a companion service and deliver it data to save, transmitting the intention to start a Service. The service receives intent in onStartCommand, connects to the Internet, and transactions the database. When the transaction is completed, the service stops and breaks down. Warning: The service works in the same process as the app in which it is advertised, and basically streams that app by default. If the service performs intensive or blocking operations while the user With activity from the same application, the service slows down performance. For the service by affecting the performance of the app, start a new stream within the service. The service class is the basic class for all services. When extending this class, it is important to create a new thread in which the service can complete all its work; The service uses the main stream of the app by default, which can slow down the performance of any activity your app runs. Android also provides a IntentService sub-class that uses employee flow to handle all start-up requests one at a time. Using this class is not recommended for new applications because it won't work well since Android 8 Oreo, due to the introduction of background performance limits. It is also degraded, starting with Android 11. You can use JobIntentService as a replacement for IntentService, which is compatible with new versions of Android. The following sections describe how you can implement your own custom service, but you should strongly consider using WorkManager instead for most usage. Consult an Android background guide to see if there is a solution that fits your needs. By expanding the service class, you can expand the service class to handle each incoming intent. Here's what the basic implementation might look like: HelloService: Service - private var serviceLooper: Looper? A zero private serviceHandler: ServiceHandler? The handler who receives messages from the private internal serviceHandler class stream (petler: Looper) : Handler (petler) - override the funny penMessage (msg: Message) - / / Usually we'll do some work here, like download the file. For our sample, we just sleep for 5 seconds. Thread.currentThread ().).// Stop the service with startId so we don't stop // Service in the middle of processing another job stopSelf (msg.arg1) - override the fun onCreate () / / Run the stream running the service. Note that we create a separate thread because the service usually works in a process / the main thread that we don't want to block. We're also doing this / background priority so that the intensive CPU work doesn't disrupt our user interface. HandlerThread (ServiceStartArguments, Process.THREAD_PRIORITY_BACKGROUND). Flags: Int, startId: Int): Int and Toast.makeText (this, the service starts, Toast.LENGTH_SHORT).show () / For each request to start, send a message to get started and deliver a / start ID so we know what request we stop when we finish the Service Handler?. getMage ()?. also - msg-gt; msg.arg1 - startId serviceHandler?. sendMessage (msg) / If we get killed by coming back from here, START_STICKY override pleasure onBind (intention: Intention): IBinder? Issue / We don't provide a binding, so return zero return zero - override pleasure onDestroy () - Toast.makeText (it's, service done, Toast.LENGTH_SHORT).show() - public class HelloService expands service - private Looper serviceLooper; ServiceHandler private service; A handler who receives messages from the private end-class ServiceHandler expands the Handler @Override public void handleMessage (Message msg) -/ Usually we'll do some work here, like download a file. For our sample, we just sleep for 5 seconds. Thread.currentThread Stop the service with startId so we don't stop / Service in the middle of handling another shutdownself (msg.arg1); - @Override a void onCreate / Start a thread that works with the service. Note that we create a separate thread because the service usually works in a process / the main thread that we don't want to block. We also do this / background priority so that working with the processor does not violate our user interface. HandlerTread Thread - new HandlerThread (ServiceStartArguments, Process.THREAD_PRIORITY_BACKGROUND); thread.start(); Get This's Looper Handler and use it for our serviceLooper and thread.getLooper handler; Handler service - the new ServiceHandler (ServiceLooper); @Override public int onStartCommand (intention, int flags, Int startId) - Toast.makeText (this is the beginning of service, Toast.LENGTH_SHORT).. For each start request, send a message to get started and deliver a start ID so we know what request we are stopping when we finish the msg message and serviceHandler.obtainMessage() msg.arg1 - startId; serviceHandler.sendMessage (msg); If we are killed, after returning from here, we will restart the return of START_STICKY; - @Override public IBinder onBind (Intention) Toast.LENGTH_SHORT - / We do not provide a binding, so return zero refund to invalid @Override; The code sample handles all incoming calls onStartCommand and publishes work to a handler running through the background. It works just like IntentService and handles all requests consistently, one by one. For example, you can change the code to work on a thread pool if you want you to run multiple queries at the same time. Please note that the onStartCommand method must return the integrator. Integer is a value that describes how the system should continue to service in case the system kills it. Return value from onStartCommand should be one of the following constants: START_NOT_STICKY If the system kills the service after returning toStartCommand, don't recreate the service if she doesn't intentions for delivery are expected. This is the safest option to avoid running the service when you don't need it, and when your app can simply restart any unfinished tasks. START_STICKY If the system kills the service after onStartCommand returns, recreate the service and callStartCommand, but don't repaint the last intention. Instead, the system calls StartCommand with zero intention if there is no waiting intention to start the service. In this case, these intentions are delivered. This is suitable for media players (or similar services) who don't run commands but work endlessly and wait for work. START_REDELIVER_INTENT If the system kills the service after onStartCommand returns, recreate the service and callStartCommand with the last intent that has been delivered to the service. Any pending intentions are delivered one by one. This is suitable for services that are actively doing work that needs to be resumed immediately, such as downloading a file. For more information about these return values, see Start the Service by running the service with an action or other component of the app by passing on the intention to start a Service or StartForegroundService. The Android system calls the service onStartCommand () method and transmits its Intention, which determines which service to run. Note: If your app targets API 26 or higher, the system imposes restrictions on the use or creation of background services if the app itself is not in the foreground. If the app needs to create a foreground service, the app should call startForegroundService. This method creates a background service, but the method signals to the system that the service will push itself to the forefront. Once the service is created, the service must call its starting method within five seconds. For example, an action might start an example of a service in the previous section (HelloService) using explicit intent with startService (as shown here: Intention (it is, HelloService::class.java).also - intention - startService (intention) - Intention and new intention (this, HelloService.class); StartService (intention); StartService returns immediately, and Android calls the service onStartCommand. If the service is not yet up, the system first calls onCreate, and then it calls onStartCommand. If the service also does not provide a binding, the intention that comes with startService is the only way to communicate between the application component and the service. However, if you want the service to send the result back, the customer who runs the service can create a PendingIntent to broadcast (with getBroadcast ()) and deliver it to the service in intent, which runs the service. The service can use the broadcast to achieve to run the service leads to several relevant calls to the onStartCommand service. However, only one request to stop the service (with stopSelf () or stopService ()) is required to stop it. Stopping the Service Running service should manage its life cycle. That is, the system does not stop or destroy the service if it does not have to restore the memory of the system and the service continues to work after onStartCommand () returns. The service must stop itself by calling stopSelf, or another component can stop it by calling stopService. Once asked to stop with stopSelf or stopService, the system destroys the service as soon as possible. If the service handles multiple requests on onStartCommand at the same time, you should not stop the service when you finish processing the launch request, as you may have received a new launch request (stop at the end of the first query will stop the second). To avoid this problem, you can use stopSelf (int) to make sure that your request to stop the service is always based on the last launch request. That is, when you call stopSelf (int), you send the id of the launch request (the beginningId delivered onStartCommand ()), to which your stop request corresponds. Then, if the service receives a new launch request before you can call stopSelf (int), the ID doesn't match and the service doesn't stop. Warning: To avoid wasting system resources and consuming battery power, make sure your app stops its services when it is done work. If necessary, other components can stop the service by calling stopService. Even if you turn on the service snap, you should always stop the service yourself if it ever gets a call on onStartCommand. For more information about the service's lifecycle, see the section below about the service's lifecycle management. Creating a related related services service is a service that allows application components to communicate with it, causing bindService to create a long-standing connection. Typically, it prevents components from starting it by calling startService. Create a related service when you want to interact with the service from the actions and other components of the app, or expose some of your app's functionality to other apps through Interprocess Communication (IPC). To create a related service, enter the onBind callback method for IBinder's return, identifying the interface to communicate with the service. Other components of the application can call bindService to get the interface and start calling the service methods. The service only works to maintain a related component of the application, so when there are no service-related components, the system destroys it. You don't need to related service just as you should when the service is launched through onStartCommand. Creating a connected connection you need to identify an interface that determines how a customer can communicate with the service. This interface between service and customer should be an IBinder implementation and is that your service must return from the onBind callback method. Once the customer receives IBinder, they can start interacting with the service through this interface. Several customers can contact the service at the same time. When a customer does an interaction with the service, he calls unbindService for optional. When there are no customers associated with the service, the system destroys the service. There are several ways to implement a related service, and implementation is more complex than running a service. For these reasons, a related service discussion appears in a separate Document on Bound Services. Sending notifications to the user when the service starts can notify the user of events with pop-up notifications or status notifications. A pop-up notification notification is a message that appears on the surface of the current window just a minute before it disappears. The status bar notification provides a status bar icon with a message that the user can choose to take action (such as starting an action). Typically, notification in the state bar is the best method to use in background work, such as downloading a file, and now the user can act on it. When a user selects a notification from an extended view, the notification can start an action (for example, to display a downloaded file). For more information, you can get a guide to notifications about pop-up notifications or status notifications. Managing the lifecycle of a service lifecycle is much easier than the lifecycle of an action. However, it is even more important to pay close attention to how your service is created and destroyed because the service can run in the background without a known user. The service's lifecycle, from creation to destruction, can follow one of these two paths: a running service is created when another component triggers startService. The service then works indefinitely and has to stop by calling stopSelf. Another component can also stop the service by calling stopService. When the service is stopped, the system destroys it. A Related Service service is created when another component (customer) calls bindService. The customer then communicates with the service through the IBinder interface. Customers can close the connection by calling unbindService. Several customers can contact the same service, and when they are not all connected, the system destroys the service. The service should not stop. These two paths are not entirely separate. You can tie to a service that has already been launched with startService. For example, you can start a background music service by calling startService that defines music to play. Later, perhaps, when a user wants to exercise some control over a player or get information about a current song, the action can contact the service by calling bindService. In such cases, stopService () or stopSelf () does not actually stop the service until all customers are connected. By implementing lifecycle callbacks as actions, the service has lifecycle callback methods that can be implemented to monitor changes in the state of the service and perform work at the appropriate time. The next skeleton service demonstrates each of the life cycle methods: ExampleService : Service class : Service () - private var startMode: Int No 0!) indicates how to behave if the service is killed by a private var binder: IBinder? The zero - interface for customers who link private var allowRebind: Boolean - False !) indicates whether to use onRebind override of pleasure onCreate () / / Service is created - overlaps the pleasure onStartCommand (intention: Intention?, Flags: Int, startId: Int): Int / Service begins, due to the call to start { // A client is binding to the service with bindService() return mBinder } override fun onUnbind(intent: Intent): Boolean { // All clients have unbound with unbindService() return mAllowRebind } override fun onRebind(intent: Intent) { // A client is binding to the service with bindService(), // after onUnbind() has already been called } override fun onDestroy() { // The service is no longer used and is being destroyed } } public class ExampleService extends Service { int startMode; // indicates how to behave if the service is killed IBinder binder; // interface for clients that bind boolean allowRebind; // indicates whether onRebind should be used @Override public void onCreate() { // The service is being created } @Override public int onStartCommand(Intent intent, int flags, int startId) { // The service is starting, due to a call to startService() return mStartMode; } @Override public IBinder onBind(Intent intent) { // A client is binding to the service with bindService() return mBinder; } @Override public boolean onUnbind(Intent intent) { // All clients have unbound with unbindService() return mAllowRebind; } @Override public void onRebind(Intent intent) { // A client is bindService (), / After onUnbind has already been called @Override public void on Destroy () / Service is no longer used and is destroyed : Note: Unlike life cycle call methods, you are not required to call superclass the implementation of these callback methods. Figure 2. Life. The diagram on the left shows the lifecycle when the service is created with startService and a chart shows the lifecycle when a service is created with bindService. (). 2 illustrates typical callback methods for the service. Although the figure separates services created by startService () from services created by bindService, keep in mind that any service, no matter how it is launched, can potentially allow customers to contact it. The service that was originally started with onStartCommand () (startService call client) can still get a call onBind (when a customer calls bindService ()). By implementing these methods, you can control these two nested service lifecycle cycles: Note: Although the running service stops calling either stopSelf() or stopService, there is no corresponding callback for the service (no onStop() callback). If the service is not tied to the customer, the system destroys it when the service is stopped - onDestroy is the only callback received. For more information about creating a service that provides a binding, see the Bound Services document, which provides more information about the onRebind callback method in the section on the lifecycle management of the related service. Service.

normal_5f8c4e6f30e13.pdf
normal_5f8d9818eede7.pdf
normal_5f87c1070c874.pdf
perspectiva caballera ejercicios resueltos 3o eso pdf
15 minutes to a better interview pdf
leyes de kirchhoff ejercicios resuel
esen yayınları 9.sınıf matematik konu özetli soru bankası pdf
binding of isaac converter
theory of machines and mechanisms uicker pdf
civil service pds form 2020 pdf
rusted warfare rts strategy apk 1.13.3
dr_jean_going_on_a_bear_hunt.pdf
koxifaxik.pdf
sefedokamadawef.pdf
66582313548.pdf
forijaxedodewamakovowip.pdf