Android Interface Definition Language (AIDL)

The Android Interface Definition Language (AIDL) is similar to other IDLs you might have worked with. It allows you to define the programming interface that both the client and service agree upon in order to communicate with each other using interprocess communication (IPC). On Android, one process cannot normally access the memory of another process. So to talk, they need to decompose their objects into primitives that the operating system can understand, and marshall the objects across that boundary for you. The code to do that marshalling is tedious to write, so Android handles it for you with AIDL.

Note: Using AIDL is necessary only if you allow clients from different applications to access your service for IPC and want to handle multithreading in your service. If you do not need to perform concurrent IPC across different applications, you should create your interface by implementing a Binder or, if you want to perform IPC, but do not need to handle multithreading, implement your interface using a Messenger. Regardless, be sure that you understand Bound Services before implementing an AIDL.

Before you begin designing your AIDL interface, be aware that calls to an AIDL interface are direct function calls. You should not make assumptions about the thread in which the call occurs. What happens is different depending on whether the call is from a thread in the local process or a remote process. Specifically:

- Calls made from the local process are executed in the same thread that is making the call. If this is your main UI thread, that thread continues to execute in the AIDL interface. If it is another thread, that is the one that executes your code in the service. Thus, if only local threads are accessing the service, you can completely control which threads are executing in it (but if that is the case, then you shouldn't be using AIDL at all, but should instead create the interface by implementing a Binder).
- Calls from a remote process are dispatched from a thread pool the platform maintains inside of your own process. You must be prepared for incoming calls from unknown threads, with multiple calls happening at the same time. In other words, an implementation of an AIDL interface must be completely thread-safe. Calls made from one thread on the same remote object arrive in order on the receiver end.
- The oneway keyword modifies the behavior of remote calls. When used, a remote call does not block; it simply sends the transaction data and immediately returns. The implementation of the interface eventually receives this as a regular call from the Binder thread pool as a normal remote call. If oneway is used with a local call, there is no impact and the call is still synchronous.

Defining an AIDL interface

You must define your AIDL interface in an .aidl file using the Java programming language syntax, then save it in the source code (in the src/ directory) of both the application hosting the service and any other application that binds to the service.

When you build each application that contains the .aidl file, the Android SDK tools generate an IBinder interface based on the .aidl file and save it in the project's gen/ directory. The service must implement the IBinder interface as appropriate. The client applications can then bind to the service and call methods from the IBinder to perform IPC.

To create a bounded service using AIDL, follow these steps:

Create the .aidl file This file defines the programming interface with method signatures.

Implement the interface The Android SDK tools generate an interface in the Java programming language, based on your .aidl file. This interface has an inner abstract class named Stub that extends Binder and implements methods from your AIDL interface. You must extend the Stub class and implement the methods.

Expose the interface to clients Implement a Service and override onBind() to return your implementation of the Stub class.

Caution: Any changes that you make to your AIDL interface after your first release must remain backward compatible in order to avoid breaking other applications that use your service. That is, because your .aidl file must be copied to other applications in order for them to access your service's interface, you must maintain support for the original interface.

1. Create the .aidl file

AIDL uses a simple syntax that lets you declare an interface with one or more methods that can take parameters and return values. The parameters and return values can be of any type, even other AIDL-generated interfaces.

You must construct the .aidl file using the Java programming language. Each .aidl file must define a single interface and requires only the interface declaration and method signatures.

By default, AIDL supports the following data types:

- All primitive types in the Java programming language (such as int, long, char, boolean, and so on)
- Arrays of primitive types such as int[]

String
CharSequence
List All elements in the List must be one of the supported data types in this list or one of the other AIDL-generated interfaces or parcelables you've declared. A List may optionally be used as a parameterized type class (for example, List). The actual concrete class that the other side receives is always an ArrayList, although the method is generated to use the List interface.

Map All elements in the Map must be one of the supported data types in this list or one of the other AIDL-generated interfaces or parcelables you've declared. Parameterized type maps, (such as those of the form Map) are not supported. The actual concrete class that the other side receives is always a HashMap, although the method is generated to use the Map interface. Consider using a Bundle as an alternative to Map.

You must include an import statement for each additional type not listed above, even if they are defined in the same package as your interface.

When defining your service interface, be aware that:

- Methods can take zero or more parameters, and return a value or void.
- All non-primitive parameters require a directional tag indicating which way the data goes. Either in, out, or inout (see the example below). Primitives, String, IBinder, and AIDL-generated interfaces are in by default, and cannot be otherwise.

Caution: You should limit the direction to what is truly needed, because marshalling parameters is expensive.

- All code comments included in the .aidl file are included in the generated IBinder interface (except for comments before the import and package statements). https://android-tech.net/
- String and int constants can be defined in the AIDL interface. For example: const int VERSION = 1;.
- Method calls are dispatched by a transact() code, which normally is based on a method index in the interface. Because this makes versioning difficult, you could manually assign the transaction code to a method: void method() = 10;.
- Nullable arguments and return types must be annotated using @nullable.

Here is an example .aidl file:

Simply save your .aidl file in your project's src/ directory and when you build your application, the SDK tools generate the IBinder interface file in your project's gen/ directory. The generated file name matches the .aidl file name, but with a .java extension (for example, IRemoteService.aidl results in IRemoteService.java).

If you use Android Studio, the incremental build generates the binder class almost immediately. If you do not use Android Studio, then the Gradle tool generates the binder class next time you build your application-you should build your project with gradle assembleDebug (or gradle assembleRelease) as soon as you're finished writing the .aidl file, so that your code can link against the generated class.

2. Implement the interface

When you build your application, the Android SDK tools generate a .java interface file named after your .aidl file. The generated interface includes a subclass named Stub that is an

abstract implementation of its parent interface (for example, YourInterface.Stub) and declares all the methods from the .aidl file.

Note: Stub also defines a few helper methods, most notably asInterface(), which takes an IBinder (usually the one passed to a client's onServiceConnected() callback method) and returns an instance of the stub interface. See the section Calling an IPC Method for more details on how to make this cast.

To implement the interface generated from the .aidl, extend the generated Binder interface (for example, YourInterface.Stub) and implement the methods inherited from the .aidl file.

Here is an example implementation of an interface called IRemoteService (defined by the IRemoteService.aidl example, above) using an anonymous instance:

Now the binder is an instance of the Stub class (a Binder), which defines the RPC interface for the service. In the next step, this instance is exposed to clients so they can interact with the service.

There are a few rules you should be aware of when implementing your AIDL interface:

- Incoming calls are not guaranteed to be executed on the main thread, so you need to think about multithreading from the start and properly build your service to be thread-safe.
- By default, RPC calls are synchronous. If you know that the service takes more than a few milliseconds to complete a request, you should not call it from the activity's main thread, because it might hang the application (Android might display an "Application is Not Responding" dialog)-you should usually call them from a separate thread in the client.
- Only the exception types listed under the reference documentation for Parcel.writeException() are sent back to the caller.

3. Expose the interface to clients

Once you've implemented the interface for your service, you need to expose it to clients so they can bind to it. To expose the interface for your service, extend Service and implement onBind() to return an instance of your class that implements the generated Stub (as discussed in the previous section). Here's an example service that exposes the IRemoteService example interface to clients.

Now, when a client (such as an activity) calls bindService() to connect to this service, the client's onServiceConnected() callback receives the binder instance returned by the service's onBind() method.

The client must also have access to the interface class, so if the client and service are in separate applications, then the client's application must have a copy of the .aidl file in its src/ directory (which generates the android.os.Binder interface-providing the client access to the AIDL methods).

When the client receives the IBinder in the onServiceConnected() callback, it must call YourServiceInterface.Stub.asInterface(service) to cast the returned parameter to YourServiceInterface type. For example:

For more sample code, see the RemoteService.java class in ApiDemos.

Passing objects over IPC

In Android 10 (API level 29), you can define Parcelable objects directly in AIDL. Types which are supported as AIDL interface arguments and other parcelables are also supported here. This avoids the additional work to manually write marshalling code and a custom class. However, this will also create a bare struct. If custom accessors or other functionality is desired, Parcelable should be implemented instead.

The code sample above automatically generates a Java class with integer fields left, top, right, and bottom. All relevant marshalling code is automatically implemented, and the object can be used directly without having to add any implementation.

You can also send a custom class from one process to another through an IPC interface. However, you must ensure that the code for your class is available to the other side of the IPC channel and your class must support the Parcelable interface. Supporting the Parcelable interface is important because it allows the Android system to decompose objects into primitives that can be marshalled across processes.

To create a custom class that supports the Parcelable protocol, you must do the following:

1. Make your class implement the Parcelable interface.
2. Implement writeToParcel, which takes the current state of the object and writes it to a Parcel.
3. Add a static field called CREATOR to your class which is an object implementing the Parcelable.Creator interface.
4. Finally, create an .aidl file that declares your parcelable class (as shown for the Rect.aidl file, below). If you are using a custom build process, do not add the .aidl file to your build. Similar to a header file in the C language, this .aidl file isn't compiled.

AIDL uses these methods and fields in the code it generates to marshall and unmarshall your objects.

For example, here is a Rect.aidl file to create a Rect class that's parcelable:

And here is an example of how the Rect class implements the Parcelable protocol.

The marshalling in the Rect class is pretty simple. Take a look at the other methods on Parcel to see the other kinds of values you can write to a Parcel.

Warning: Don't forget the security implications of receiving data from other processes. In this case, the Rect reads four numbers from the Parcel, but it is up to you to ensure that these are within the acceptable range of values for whatever the caller is trying to do. See Security and Permissions for more information about how to keep your application secure from malware.

Methods with Bundle arguments containing Parcelables

if you have an .aidl file:

Calling an IPC method

Here are the steps a calling class must take to call a remote interface defined with AIDL:

1. Include the .aidl file in the project src/ directory.
2. Declare an instance of the IBinder interface (generated based on the AIDL).
3. Implement ServiceConnection.
4. Call Context.bindService(), passing in your ServiceConnection implementation.
5. In your implementation of onServiceConnected(), you will receive an IBinder instance (called service). Call YourInterfaceName.Stub.asInterface((IBinder)service) to cast the returned parameter to YourInterface type.
6. Call the methods that you defined on your interface. You should always trap DeadObjectException exceptions, which are thrown when the connection has broken. You should also trap SecurityException exceptions, which are thrown when the two processes involved in the IPC method call have conflicting AIDL definitions.
7. To disconnect, call Context.unbindService() with the instance of your interface.

A few comments on calling an IPC service:

- Objects are reference counted across processes.
- You can send anonymous objects as method arguments.