Building Connected Apps

Overview

Connected apps is a framework-backed mechanism of giving your app access to content in both work and personal profiles to display to users in a single user interface.

What can I expect? How difficult is this to build?

The connected apps SDK makes it as easy as possible to perform and test cross-profile behaviour or data access, once you have the appropriate permission. Normally, the cross-profile calls themselves are quite straightforward, so you should be able to estimate the complexity and engineering cost from your specific product goals and other desired changes within your app itself.

Principles

The general flow of data through your app should not change to accommodate becoming connected. At a high-level, do not think "send a message to the other profile to tell them that X" or "get data from the other profile to Y". Instead, continue to think "get emails", "delete event", or "register listener". A device only ever has one APK per package name, so each profile will always have an identical version of the app. At a single point in each data flow stack, the connected apps SDK can be used to route the call to the appropriate profile(s).

Permissions

Firstly, determine which category your app falls into: **full-consent**, **pre-granted**, or **cross-user**. This will be referenced throughout this document.

If your privileged app already has the `INTERACT_ACROSS_USERS` permission then you are **cross-user**. If you are listed in the `cross_profile_apps.xml` file, you are **pre-granted**. Otherwise, as with the vast majority of apps, you are **full-consent**.

FYI - product caveats

- Work data will not be accessible when the work profile is off.
- The connected apps SDK is only supported from Oreo-onwards, which encompasses the vast majority of work profiles. As discussed in the next section, this requirement will actually be from Android 11 onwards for most apps.

The following points apply to full-consent and pre-granted apps only:

- The permission is always user-revocable, so assume that you have to build for the case where the apps are not connected.
- Connected apps is only supported from Android 11 onwards.
- Apps using this permission will need to be included in a Play Store allowlist.

The following points apply to full-consent apps only:

- The default state will be off. You will need to send the user through the consent flow to request the permission.
- The IT admin will need to consent to your app as well, or the user won't be able to give the permission.
- The app will need to be installed in both profiles. The consent flow already prompts the user to install in the other profile if they haven't already.

Getting Started

Overview

This section is designed to give engineers a 5-minute overview of how cross-profile calls work with the SDK and how they can be tested. Don't try to build anything yet - this isn't designed to be used as a reference or a guide, but just as an introduction.

Custom profile-aware classes and methods

The connected apps SDK allows you to annotate your own classes and methods as cross-profile. This generates classes and methods that allow you to execute the annotated method on any profile.

For example, consider the following class, with SDK annotation added:

```
public class CalendarDatabase {

  @CrossProfile // SDK annotation
  public void deleteEvent(Event event, Account account) {
    // complex logic with database calls
  }
}
```

This generates a class prefixed with Profile, allowing you to call this API on the profile of your choice. For example:

```
profileCalendarDatabase.work().deleteEvent(event, account);
```

More Complex Examples

More realistically, your classes and methods will be more complex. For example, your existing API could use `ListenableFuture` return types and you might need to combine results from both profiles. Consider this example:

```
public class CalendarDatabase {

  @CrossProfile // SDK annotation
  public ListenableFuture<Collection<Event>> getEvents() {
    // complex logic with database calls
  }
}
```

```
// Merge results from both profiles into a set
profileCalendarDatabase.both()
    .getEvents()
    .transform((Map<Profile, Collection<Event>> events) -> {
      return events.values()
          .stream()
          .flatMap(Collection::stream)
          .collect(Collectors.toSet());
    }, directExecutor());
```

These generated classes and methods work as expected with full type safety and IDE code completion.

Each return and parameter type of your annotated APIs must be supported by the SDK, but it fully supports nesting and generics of lists, sets, arrays, primitives, any parcelable type, and any serializable type, in addition to `ListenableFuture`, `Optional`, and protos. It's also possible for you to add support for types not natively supported by the SDK. As an extreme example, it would seamlessly support `ListenableFuture<List<Map<CustomProto, CustomParcelableType[]>>>`.

Testing

The SDK is designed to make unit testing as easy as possible. For each generated `Profile` class, there is a corresponding `FakeProfile` class that you can provide work and personal instances to. For example:

```
// Create an instance of the SDK-generated fake connector class. This
// class allows you to control the availability of the profiles, which
// profile you are currently running on, etc.
private final FakeCrossProfileConnector connector =
    new FakeCrossProfileConnector();

// Create an instance of your real/fake/mock class for both profiles.
private final CalendarDatabase personalCalendarDatabase =
    new FakeCalendarDatabase();
private final CalendarDatabase workCalendarDatabase =
    new FakeCalendarDatabase();

// Create an instance of the SDK-generated fake profile-aware class.
private final FakeProfileCalendarDatabase profileCalendarDatabase =
    FakeProfileCalendarDatabase.builder()
        .personal(personalCalendarDatabase)
        .work(workCalendarDatabase)
        .connector(connector)
        .build();

// Pass profileCalendarDatabase into your classes under test, or set
// Dagger up to inject the fake automatically.
```

Hello World

This section shows you how to create a cross-profile hello world call within your app. This will give you familiarity with the SDK and an initial implementation to build on top of. You are recommended to actively follow along by developing in your app.

Ensure that you have a test device with a work profile to test on

The easiest way to do this is with an app called TestDPC. This app allows you to mimic being an IT admin, including creating and managing a work profile. Download the TestDPC apk from here, then install it with adb and open the 'set up...' icon to create the work profile. When installing your app, it will normally automatically install in both profiles. If you ever need to install it explicitly into the work profile, you can use the `--user` argument with adb install.

Ensure your application is configured appropriately

Enable java 8 support and ensure your minSdk is at least 19.

Add gradle dependencies

```
dependencies {
    annotationProcessor
'com.google.android.enterprise.connectedapps:connectedapps-processor:1.0.0-alpha05'
    implementation
'com.google.android.enterprise.connectedapps:connectedapps:1.0.0-alpha05'
    implementation
'com.google.android.enterprise.connectedapps:connectedapps-annotations:1.0.0-alpha0
5'
}
```

We use Guava in this example. This is not a requirement of using the SDK, but to follow along with the hello world you should also add `api("com.google.guava:guava:29.0-android")`

Create a new class that will contain your test cross-profile call

To make this more useful later, you should put it in the same package that you would like your real cross-profile calls to go in.

```
public class HelloWorld {

  @CrossProfile
  public ListenableFuture<String> helloWorld() {
    return Futures.immediateFuture("Hello world");
  }
}
```

If you can't depend on Guava for Futures support, just follow along for now and then see the final step which tells you which changes to make.

Make a cross-profile call

You could do this in a class that will need to make real cross-profile calls later.

```
// TODO: inject/pass these into the class later instead.
CrossProfileConnector crossProfileConnector =
    CrossProfileConnector.builder(this).build();
ProfileHelloWorld profileHelloWorld =
    ProfileHelloWorld.create(crossProfileConnector);

ListenableFuture<Map<Profile, String>> resultsFuture =
    profileHelloWorld.both().helloWorld();

FluentFuture.from(resultsFuture)
    .addCallback(new FutureCallback<Map<Profile, String>>() {
      @Override
      public void onSuccess(Map<Profile, String> results) {
        for (Profile profile : results.keySet()) {
          Log.w("tag", "CROSS_PROFILE profile: " + profile.asInt()
              + "; result: " + results.get(profile));
        }
      }
      @Override
      public void onFailure(Throwable t) {
        Log.e(TAG, "Failed to say hello world on both profiles", t);
```

```
    }
  }, directExecutor());
```

Provide the instance to the SDK

Of course, the `helloWorld` method call above is on a generated class and not the real one. Often, your real classes are singletons or other complex classes dependent on dependency injection frameworks such as Dagger.

To allow the logic to be called on the real class on the other profile, each custom `@CrossProfile` class must have a corresponding provider method in a `@CrossProfileProvider` class. Create this class.

```
public class HelloWorldProvider {

  @CrossProfileProvider
  public HelloWorld getHelloWorld() {
    return new HelloWorld();
  }
}
```

Additional wiring

The code generation required to support custom classes and methods requires a small amount of additional wiring. This is to handle the complexities of having many build targets and visibility requirements at scale.

Firstly, annotate an existing or new high-level class with `@CrossProfileConfiguration`, pointing to your provider classes.

```
@CrossProfileConfiguration(providers = HelloWorldProvider.class)
abstract class HelloWorldConfiguration {}
```

Secondly, add the automatically-generated service to your manifest, inside the <application> tag. This may not resolve until you build your project.

```
<service
android:name="com.google.android.enterprise.connectedapps.CrossProfileConnector_Ser
vice"
android:exported="false"/>
```

Finally, for development purposes, give yourself the INTERACT_ACROSS_USERS permission. If you don't have it already, you won't be able to keep this in production, but it's the easiest way to get started. Firstly, add it to your manifest as follows:

```
<uses-permission
      android:name="android.permission.INTERACT_ACROSS_USERS"
      tools:ignore="ProtectedPermissions"/>
```

Then, you can grant yourself the permission from the command line with adb as follows (if your app does not already have it):

```
adb shell pm grant <your package> android.permission.INTERACT_ACROSS_USERS
```

If you were not able to depend on Guava for Futures, you will need to make a few changes. Firstly, simply return the "Hello World" string directly and print the results of the cross-profile call out. Secondly, since these calls are now synchronous, you should place your cross-profile call and printing of results inside the connection listener.

When you run the code listed under "make a cross-profile call" you should see two logs for "Hello World", one from each profile. If you only get one log, make sure you've installed your app in both profiles and have granted the permission.

Production Ready

This section assumes that you have already completed the hello world guide. It will take you through converting this into a full implementation. It often links out to the development reference sections of this document.

Switch out the logic from the hello world example for your real cross-profile logic

Feel free to keep it messy and not worry about best practices like testability yet. Make sure to check that it works!

This includes modifying the cross-profile class, the cross-profile calls, and the provider. If you need a `Context` in the provider method or the constructor of the provider class, you can add one as a parameter and the SDK will automatically provide it.

You will need to decide whether your calls will be asynchronous or synchronous. You should also ensure that all parameter and return types used by your cross-profile calls are supported.

Finally, add version checks. For **cross-user apps**, the SDK calls will only work on Oreo+. For everybody else, the SDK calls will only work on Android 11+.

Refactor your logic to have an architecture that you could submit

Read through each of the architectural suggestions first so you don't have to do multiple conflicting refactorings. Once you've planned out any changes, work back through them and check them off, re-building and testing after each change.

Double-check privacy/security

Double-check that you are not at risk of storing data in the wrong profile or sending data to a server (including server logging) from the wrong profile.

Consider other best practices

- Consider designating a profile as primary to simplify your calls. Think about what happens if your logic is run on either profile.
- Refresh your UI when the availability of the other profile changes.
- Review the discussion of exceptions in case you need to take any action.
- If you expect users might only use the app icon in one of the two profiles, add the `android:crossProfile = "true"` Manifest attribute. This prevents the app not directly used by the user from being delegated into lower app standby buckets.

Add tests

See testing documentation.

Consider permissions

If your app is **full-consent**, switch your permission from `INTERACT_ACROSS_USERS` to `INTERACT_ACROSS_PROFILES` and implement requesting the permission from the user. Apps declaring `INTERACT_ACROSS_PROFILES` can't be uploaded to the Play store until allow-listed, so this manifest change should be behind a flag until this is done.

If your app is **pre-granted**, you might also want to do this if you want to prompt users who previously turned it off (but be responsible - do not spam users).

Details

These sections are meant for reference and it is not required that you read them top-to-bottom.

Requesting user consent

For now, use framework APIs `CrossProfileApps#canInteractAcrossProfiles`,

`CrossProfileApps#canRequestInteractAcrossProfiles`, `CrossProfileApps#createRequestInteractAcrossProfilesIntent`, and `CrossProfileApps#ACTION_CAN_INTERACT_ACROSS_PROFILES_CHANGED`.

These APIs will be wrapped in the SDK for a more consistent API surface (e.g. avoiding UserHandle objects), but for now, you can call these directly.

The implementation is straightforward: if you can interact, go ahead. If not, but you can request, then show your user prompt/banner/tooltip/etc. If the user agrees to go to Settings, create the request intent and use Context#startActivity to send the user there. You can either use the broadcast to detect when this ability changes, or just check again when the user comes back.

To test this, you'll need to open TestDPC in your work profile, scroll to the very bottom and select to add your package name to the connected apps allowlist. This mimics the admin allowlisting your app.

Glossary

Cross Profile Configuration

A Cross Profile Configuration groups together related Cross Profile Provider Classes and provides general configuration for the cross-profile features. Typically there will be one `@CrossProfileConfiguration` annotation per codebase, but in some complex applications there may be multiple.

Profile Connector

A Connector manages connections between profiles. Typically each cross profile type will point to a specific Connector. Every cross profile type in a single configuration must use the same Connector.

Cross Profile Provider Class

A Cross Profile Provider Class groups together related Cross Profile Types.

Mediator

A mediator sits between high-level and low-level code, distributing calls to the correct profiles and merging results. This is the only code which needs to be profile-aware. This is an architectural concept rather than something built into the SDK.

Cross Profile Type

A cross profile type is a class or interface containing methods annotated `@CrossProfile`. The code in this type needs not be profile-aware and should ideally just act on its local data.

Profile Types

Profile Type	Description
Current	The profile we are currently executing on.
Other	(if it exists) The profile we are not currently executing on.
Personal	User 0, the profile on the device that cannot be switched off.
Work	Typically user 10 but may be higher, can be toggled on and off, used to contain work apps and data.
Primary	Optionally defined by the application. The profile which shows a merged view of both profiles.

Secondary	If primary is defined, secondary is the profile which is not primary.
Supplier	The suppliers to the primary profile is both profiles, the suppliers to the secondary profile is only the secondary profile itself.

Profile Identifier

A class which represents a type of profile (personal or work). These will be returned by methods which run on multiple profiles and can be used to run more code on those profiles. These can be serialised to an int for easy storage.

Architectural Best Practices

Convert your CrossProfileConnector into a singleton

Only a single instance should be used throughout the lifecycle of your application, or else you will create parallel connections. This can be done either using a dependency injection framework such as Dagger, or by using a classic Singleton pattern, either in a new class or an existing one.

Inject/pass in the generated Profile instance into your class for when you make the call, rather than creating it in the method

This allows you to pass in the automatically-generated `FakeProfile` instance in your unit tests later.

Consider the mediator pattern

This common pattern is to make one of your existing APIs (e.g. `getEvents()`) profile-aware for all of its callers. In this case, your existing API can just become a 'mediator' method/class that contains the new call to generated cross-profile code. This way, you don’t force every caller to know to make a cross-profile call - it just becomes part of your API.

Consider whether to annotate an interface method as @CrossProfile instead to avoid having to expose your implementation classes in a provider

This works nicely with dependency injection frameworks.

If you are receiving any data from a cross-profile call, consider whether to add a field referencing which profile it came from

This can be good practice since you might want to know this at the UI layer (e.g. adding a badge icon to work stuff). It also might be required if any data identifiers are no longer unique without it, such as package names.

Primary Profiles

Most of the calls in examples on this doc contain explicit instructions on which profiles to run on, including work, personal, and both.

In practice, for apps with a merged experience on only one profile, you likely want this decision to depend on the profile that you are currently running on, so there are similar convenient methods that also take this into account, to avoid your codebase being littered with if-else profile conditionals.

When creating your connector instance, you can specify which profile type is your 'primary' (e.g. 'WORK'). This enables additional options, such as the following:

```
profileCalendarDatabase.primary().getEvents();

profileCalendarDatabase.secondary().getEvents();

// Runs on all profiles if currently running on the primary, or just
// on the current profile if currently running on the secondary.
profileCalendarDatabase.suppliers().getEvents();
```

Cross Profile Types

Classes and interfaces which contain a method annotated `@CrossProfile` are referred to as Cross Profile Types.

The implementation of Cross Profile Types should not be aware of the profile they are running on. They are allowed to make calls to other methods and in general should work like they were running on a single profile. They will only have access to state in their own profile.

An example Cross Profile Type:

```
public class Calculator {
    @CrossProfile
    public int add(int a, int b) {
       return a + b;
    }
}
```

Class annotation

To provide the strongest API, you should specify the connector for each cross profile type, as so:

```
@CrossProfile(connector=MyProfileConnector.class)
public class Calculator {
    @CrossProfile
    public int add(int a, int b) {
       return a + b;
    }
}
```

This is optional but means that the generated API will be more specific on types and stricter on compile-time checking.

Interfaces

By annotating methods on an interface as `@CrossProfile` you are stating that there can be some implementation of this method which should be accessible across profiles. You can return any implementation of a Cross Profile interface in a Cross Profile Provider and by doing so you are saying that this implementation should be accessible cross-profile. You do not need to annotate the implementation classes.

Cross Profile Providers

Every Cross Profile Type must be provided by a method annotated `@CrossProfileProvider`. These methods will be called each time a cross-profile call is made, so it is recommended that you maintain singletons for each type.

Constructor

A provider must have a public constructor which takes either no arguments or a single `Context` argument.

Provider Methods

Provider methods must take either no arguments or a single `Context` argument.

Dependency Injection

If you're using a dependency injection framework such as Dagger to manage dependencies, we recommend that you have that framework create your cross profile types as you usually would, and then inject those types into your provider class. The `@CrossProfileProvider` methods can then return those injected instances.

Profile Connector

Each Cross Profile Configuration must have a single Profile Connector, which is responsible for managing the connection to the other profile.

The Profile Connector's connect method must be called before any blocking interaction with the SDK. See synchronous calls. If you are only using asynchronous calls, this does not apply to you.

Default Profile Connector

If there is only one Cross Profile Configuration in a codebase, then you can avoid creating your own Profile Connector and use `com.google.android.enterprise.connectedapps.CrossProfileConnector`. This is the default used if none is specified.

When constructing the CrossProfileConnector, you can specify some options on the builder:

Scheduled Executor Service

If you want to have control over the threads created by the SDK, use `#setScheduledExecutorService()`

Binder

If you have specific needs regarding profile binding, use `#setBinder`. This is likely only used by Device Policy Controllers.

Custom Profile Connector

You will need a custom profile connector to be able to set some configuration (using `CustomProfileConnector`) and will need one if you need multiple connectors in a single codebase (for example if you have multiple processes, we recommend one connector per process).

When creating a `ProfileConnector` it should look like:

```
@GeneratedProfileConnector
public interface MyProfileConnector extends ProfileConnector {
  public static MyProfileConnector create(Context context) {
    // Configuration can be specified on the builder
    return GeneratedMyProfileConnector.builder(context).build();
  }
}
```

You should use a single instance of your profile connector throughout your application. This can be managed by dagger or other dependency injection framework, or by having a class maintain the singleton.

When creating a custom profile connector, apply the `@CustomProfileConnector` annotation to your class to specify options:

serviceClassName

To change the name of the service generated (which should be referenced in your `AndroidManifest.xml`), use `serviceClassName=`.

primaryProfile

To specify the primary profile, use `primaryProfile=`.

availabilityRestrictions

To change the restrictions the SDK places on connections and profile availability, use `availabilityRestrictions=`.

Device Policy Controllers

If your app is a Device Policy Controller, then you must specify an instance of `DpcProfileBinder` referencing your `DeviceAdminReceiver`.

If you are implementing your own profile connector:

```
@GeneratedProfileConnector
public interface DpcProfileConnector extends ProfileConnector {
  public static DpcProfileConnector get(Context context) {
    return GeneratedDpcProfileConnector.builder(context).setBinder(new
DpcProfileBinder(new ComponentName("com.google.testdpc",
"AdminReceiver"))).build();
  }
}
```

or using the default `CrossProfileConnector`:

```
CrossProfileConnector connector =
CrossProfileConnector.builder(context).setBinder(new DpcProfileBinder(new
ComponentName("com.google.testdpc", "AdminReceiver"))).build();`
```

Cross Profile Configuration

The `@CrossProfileConfiguration` annotation is used to link together all cross profile types using a connector in order to dispatch method calls correctly. To do this, we annotate a class with `@CrossProfileConfiguration` which points to every provider, like so:

```
@CrossProfileConfiguration(providers = {TestProvider.class})
public abstract class TestApplication {
}
```

This will validate that for all Cross Profile Types they have either the same profile connector or no connector specified.

serviceSuperclass

By default, the generated service will use `android.app.Service` as the superclass. If you need a different class (which itself must be a subclass of android.app.Service) to be the superclass, then specify `serviceSuperclass=`.

serviceClass

If specified, then no service will be generated. This must match the `serviceClassName` in the profile connector you are using. Your custom service should dispatch calls using the generated `_Dispatcher` class as such:

```
public final class TestProfileConnector_Service extends Service {
  private Stub binder = new Stub() {
    private final TestProfileConnector_Service_Dispatcher dispatcher = new
TestProfileConnector_Service_Dispatcher();
```

```
    @Override
    public void prepareCall(long callId, int blockId, int numBytes, byte[] params)
{
      dispatcher.prepareCall(callId, blockId, numBytes, params);
    }

    @Override
    public byte[] call(long callId, int blockId, long crossProfileTypeIdentifier,
int methodIdentifier, byte[] params,
    ICrossProfileCallback callback) {
      return dispatcher.call(callId, blockId, crossProfileTypeIdentifier,
methodIdentifier, params, callback);
    }

    @Override
    public byte[] fetchResponse(long callId, int blockId) {
      return dispatcher.fetchResponse(callId, blockId);
  };

  @Override
  public Binder onBind(Intent intent) {
    return binder;
  }
}
```

This can be used if you need to perform additional actions before or after a cross-profile call.

Connector

If you are using a connector other than the default `CrossProfileConnector` then you must specify it using `connector=`.

Visibility

Every part of your application which interacts cross-profile must be able to see your Profile Connector.

Your `@CrossProfileConfiguration` annotated class must be able to see every provider used in your application.

Synchronous Calls

The Connected Apps SDK supports synchronous (blocking) calls for cases where they are unavoidable. However, there are a number of disadvantages to using these calls (such as the potential for calls to block for a long time) so it is recommended that you **avoid synchronous calls when possible**. For using asynchronous calls see Asynchronous calls.

Connection Holders

If you are using synchronous calls, then you must ensure that there is a connection holder registered before making cross profile calls, otherwise an exception will be thrown. For more information see Connection Holders.

To add a connection holder, call `ProfileConnector#addConnectionHolder(Object)` with any object (potentially, the object instance which is making the cross-profile call). This will record that this object is making use of the connection and will attempt to make a connection. This must be called **before** any synchronous calls are made. This is a non-blocking call so it is possible that the connection won't be ready (or may not be possible) by the time you make your call, in which case the usual error handling behaviour applies.

If you lack the appropriate cross-profile permissions when you call `ProfileConnector#addConnectionHolder(Object)` or no profile is available to connect, then no error will be thrown but the connected callback will never be called. If the permission is later granted or the other profile becomes available then the connection will be made then and the callback called.

Alternatively, `ProfileConnector#connect(Object)` is a blocking method which will add the object as a connection holder and either establish a connection or throw an `UnavailableProfileException`. **This method can not be called from the UI Thread**.

Calls to `ProfileConnector#connect(Object)` and the similar `ProfileConnector#connect` return auto-closing objects which will automatically remove the connection holder once closed. This allows for usage such as:

```
try (ProfileConnectionHolder p = connector.connect()) {
  // Use the connection
}
```

Once you are finished making synchronous calls, you should call `ProfileConnector#removeConnectionHolder(Object)`. Once all connection holders are removed, the connection will be closed.

Connectivity

A connection listener can be used to be informed when the connection state changes, and `connector.utils().isConnected` can be used to determine if a connection is present. For example:

```
// Only use this if using synchronous calls instead of Futures.
crossProfileConnector.connect(this);
crossProfileConnector.registerConnectionListener(() -> {
  if (crossProfileConnector.utils().isConnected()) {
    // Make cross-profile calls.
  }
});
```

Asynchronous Calls

Every method exposed across the profile divide must be designated as blocking (synchronous) or non-blocking (asynchronous). Any method which returns an asynchronous data type (e.g. a `ListenableFuture`) or accepts a callback parameter is marked as non-blocking. All other methods are marked as blocking.

Asynchronous calls are recommended. If you must use synchronous calls see Synchronous Calls.

Callbacks

The most basic type of non-blocking call is a void method which accepts as one of its parameters an interface which contains a method to be called with the result. To make these interfaces work with the SDK, the interface must be annotated `@CrossProfileCallback`. For example:

```
@CrossProfileCallback
public interface InstallationCompleteListener {
    void installationComplete(int state);
}
```

This interface can then be used as a parameter in a `@CrossProfile` annotated method and be called as usual. For example:

```
@CrossProfile
public void install(String filename, InstallationCompleteListener callback) {
    // Do something on a separate thread and then:
    callback.installationComplete(1);
}

// In the mediator
profileInstaller.work().install(filename, (status) -> {
    // Deal with callback
}, (exception) -> {
    // Deal with possibility of profile unavailability
});
```

If this interface contains a single method, which takes either zero or one parameters, then it can also be used in calls to multiple profiles at once.

Any number of values can be passed via a callback, but the connection will only be held open for the first value. See Connection Holders for information on holding the connection open to receive more values.

Synchronous methods with callbacks

One unusual feature of using callbacks with the SDK is that you could technically write a synchronous method which uses a callback:

```
public void install(InstallationCompleteListener callback) {
    callback.installationComplete(1);
}
```

In this case, the method is actually synchronous, despite the callback. This code would execute correctly:

```
System.out.println("This prints first");
installer.install(() -> {
        System.out.println("This prints second");
});
System.out.println("This prints third");
```

However, when called using the SDK, this will not behave in the same way. There is no guarantee that the install method will have been called before “This prints third” is printed. Any uses of a method marked as asynchronous by the SDK must make no assumptions about when the method will be called.

Simple Callbacks

"Simple callbacks" are a more restrictive form of callback which allows for additional features when making cross-profile calls. Simple interfaces must contain a single method, which can take either zero or one parameters.

You can enforce that a callback interface must remain simply by specifying `simple=true` in the `@CrossProfileCallback` annotation.

Simple callbacks are usable with `.both()`, `.suppliers()`, etc.

Connection Holders

When making an asynchronous call (using either callbacks or futures) a connection holder will be added when making the call and removed when either an exception or a value is passed.

If you expect more than one result to be passed via a callback, you should manually add the callback as a connection holder:

```
MyCallback b = //...
connector.addConnectionHolder(b);

 profileMyClass.other().registerListener(b);

 // Now the connection will be held open indefinitely, once finished:
 connector.removeConnectionHolder(b);
```

This can also be used with a try-with-resources block:

```
MyCallback b = //...
try (ProfileConnectionHolder p = connector.addConnectionHolder(b)) {
  profileMyClass.other().registerListener(b);

  // Other things running while we expect results
}
```

If we make a call with a callback or future, the connection will be held open until a result is passed. If we determine that a result will not be passed, then we should remove the callback or future as a connection holder:

```
connector.removeConnectionHolder(myCallback);
connector.removeConnectionHolder(future);
```

For more information see Connection Holders

Futures

Futures are also supported natively by the SDK. Currently the only natively supported Future type is `ListenableFuture`, though custom future types can be used. To use futures you just declare a supported Future type as the return type of a cross profile method and then use it as normal.

This has the same "unusual feature" as callbacks, where a synchronous method which returns a future (e.g. using `immediateFuture`) will behave differently when run on the current profile vs run on another profile. Any uses of a method marked as asynchronous by the SDK must make no assumptions about when the method will be called.

Threads

Do not block on the result of a cross-profile future or callback on the main thread. If you do this, then in some situations your code will block indefinitely. This is because the connection to the other profile is also established on the main thread, which will never occur if it is blocked pending a cross-profile result.

Availability

An availability listener can be used to be informed when the availability state changes, and `connector.utils().isAvailable` can be used to determine if another profile is available for use. For example:

```
crossProfileConnector.registerAvailabilityListener(() -> {
  if (crossProfileConnector.utils().isAvailable()) {
    // Show cross-profile content
  } else {
    // Hide cross-profile content
  }
});
```

Connection Holders

Connection holders are arbitrary objects which are recorded as having and interest in the cross-profile connection being established and kept alive.

By default, when making asynchronous calls, a connection holder will be added when the call starts, and removed when any result or error occurs.

Connection Holders can also be added and removed manually to exert more control over the connection. Connection holders can be added using `connector.addConnectionHolder`, and removed using `connector.removeConnectionHolder`.

When there is at least one connection holder added, the SDK will attempt to maintain a connection. When there are zero connection holders added, the connection can be closed.

You must maintain a reference to any connection holder you add - and remove it when it is no longer relevant.

Synchronous calls

Before making synchronous calls, a connection holder should be added. This can be done using any object, though you must keep track of that object so it can be removed when you no longer need to make synchronous calls.

Asynchronous calls

When making asynchronous calls connection holders will be automatically managed so that the connection is open between the call and the first response or error. If you need the connection to survive beyond this (e.g. to receive multiple responses using a single callback) you should add the callback itself as a connection holder, and remove it once you no longer need to receive further data.

Error Handling

By default, any calls made to the other profile when the other profile is not available will result in an `UnavailableProfileException` being thrown (or passed into the Future, or error callback for an async call).

To avoid this, developers can use `#both()` or `#suppliers()` and write their code to deal with any number of entries in the resulting list (this will be 1 if the other profile is unavailable, or 2 if it is available).

Exceptions

Any unchecked exceptions which happen after a call to the current profile will be propagated as usual. This applies regardless of the method used to make the call (`#current()`, `#personal`, `#both`, etc.).

Unchecked exceptions which happen after a call to the other profile will result in a `ProfileRuntimeException` being thrown with the original exception as the cause. This applies regardless of the method used to make the call (`#other()`, `#personal`, `#both`, etc.).

Checked exceptions are not currently supported in Cross Profile methods.

ifAvailable

As an alternative to catching and dealing with `UnavailableProfileException` instances, you can use the `.ifAvailable()` method to provide a default value which will be returned instead of throwing an `UnavailableProfileException`.

For example:

```
profileNotesDatabase.other().ifAvailable().getNumberOfNotes(/* defaultValue= */ 0);
```

Testing

To make your code testable, you should be injecting instances of your profile connector to any code which uses it (to check for profile availability, to manually

connect, etc.). You should also be injecting instances of your profile aware types where they are used.

We provide fakes of your connector and types which can be used in tests.

First, add the test dependencies:

```
    testAnnotationProcessor
'com.google.android.enterprise.connectedapps:connectedapps-processor:1.0.0-alpha05'
    testCompileOnly
'com.google.android.enterprise.connectedapps:connectedapps-testing-annotations:1.0.
0-alpha05'
    testImplementation
'com.google.android.enterprise.connectedapps:connectedapps-testing:1.0.0-alpha05'
```

Then, annotate your test class with `@CrossProfileTest`, identifying the `@CrossProfileConfiguration` annotated class to be tested:

```
@CrossProfileTest(configuration = MyApplication.class)
@RunWith(RobolectricTestRunner.class)
public class NotesMediatorTest {

}
```

This will cause the generation of fakes for all types and connectors used in the configuration.

Create instances of those fakes in your test:

```
private final FakeCrossProfileConnector connector = new
FakeCrossProfileConnector();
private final NotesManager personalNotesManager = new NotesManager(); //
real/mock/fake
private final NotesManager workNotesManager = new NotesManager(); // real/mock/fake

private final FakeProfileNotesManager profileNotesManager =
    FakeProfileNotesManager.builder()
        .personal(personalNotesManager)
        .work(workNotesManager)
        .connector(connector)
        .build();
```

Set up the profile state:

```
connector.setRunningOnProfile(PERSONAL);
connector.createWorkProfile();
connector.turnOffWorkProfile();
```

Pass the fake connector and cross profile class into your code under test and then make calls.

Calls will be routed to the correct target - and exceptions will be thrown when making calls to disconnected/unavailable profiles.

Types

Supported Types

The following types are supported with no extra effort on your part. These can be used as either arguments or return types for all cross-profile calls.

- Primitives (`byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`)
- Boxed Primitives (`java.lang.Byte`, `java.lang.Short`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Float`, `java.lang.Double`, `java.lang.Character`, `java.lang.Boolean`, `java.lang.Void`)
- `java.lang.String`
- Anything which implements `android.os.Parcelable`
- Anything which implements `java.io.Serializable`
- Single-dimension non-primitive arrays
- `java.util.Optional`
- `java.util.Collection`
- `java.util.List`
- `java.util.Map`
- `java.util.Set`
- `android.util.Pair`
- `com.google.common.collect.ImmutableMap`

Any supported generic types (for example `java.util.Collection`) may have any supported type as their type parameter. For example,

`java.util.Collection<java.util.Map<java.lang.String, MySerializableType[]>>` is a valid type.

Futures

The following types are supported only as return types:

- `com.google.common.util.concurrent.ListenableFuture`

Custom Parcelable Wrappers

If your type is not in the above list, first consider if it can be made to correctly implement either `android.os.Parcelable` or `java.io.Serializable`. If it cannot then see parcelable wrappers to add support for your type.

Custom Future Wrappers

If you wish to use a future type which is not in the above list, see future wrappers to add support.

Parcelable Wrappers

Parcelable Wrappers are the way that the SDK adds support for unparcelable types which cannot be modified. The SDK includes wrappers for many types but if the type you need to use is not included you must write your own.

A Parcelable Wrapper is a class designed to wrap another class and make it parcelable. It follows a defined static contract and is registered with the SDK so it can be used to convert a given type into a parcelable type, and also extract that type from the parcelable type.

In this section we will refer to the wrapped type as `T`, and the wrapper type as `W`.

Annotation

The parcelable wrapper class must be annotated `@CustomParcelableWrapper`, specifying the wrapped class as `originalType`. For example:

```
@CustomParcelableWrapper(originalType=ImmutableList.class)
```

Format

Parcelable wrappers must implement `Parcelable` correctly, and must have a static `W of(Bundler, BundlerType, T)` method which wraps the wrapped type and a non-static `T get()` method which returns the wrapped type.

The SDK will use these methods to provide seamless support for the type.

Bundler

To allow for wrapping generic types (such as lists and maps), the `of` method is passed a `Bundler` which is capable of reading (using `#readFromParcel`) and writing (using `#writeToParcel`) all supported types to a `Parcel`, and a `BundlerType` which represents the declared type to be written.

`Bundler` and `BundlerType` instances are themselves parcelable, and should be written as part of the parcelling of the parcelable wrapper, so that it can be used when reconstructing the parcelable wrapper.

If the `BundlerType` represents a generic type, the type variables can be found by calling `.typeArguments()`. Each type argument is itself a `BundlerType`.

For an example see ParcelableCustomWrapper:

```
public class CustomWrapper<F> {
  private final F value;

  public CustomWrapper(F value) {
    this.value = value;
  }

  public F value() {
    return value;
  }
}
```

```java
@CustomParcelableWrapper(originalType = CustomWrapper.class)
public class ParcelableCustomWrapper<E> implements Parcelable {

  private static final int NULL = -1;
  private static final int NOT_NULL = 1;

  private final Bundler bundler;
  private final BundlerType type;
  private final CustomWrapper<E> customWrapper;

  /**
   * Create a wrapper for a given {@link CustomWrapper}.
   *
   * <p>The passed in {@link Bundler} must be capable of bundling {@code F}.
   */
  public static <F> ParcelableCustomWrapper<F> of(
      Bundler bundler, BundlerType type, CustomWrapper<F> customWrapper) {
    return new ParcelableCustomWrapper<>(bundler, type, customWrapper);
  }

  public CustomWrapper<E> get() {
    return customWrapper;
  }

  private ParcelableCustomWrapper(
      Bundler bundler, BundlerType type, CustomWrapper<E> customWrapper) {
    if (bundler == null || type == null) {
      throw new NullPointerException();
    }
    this.bundler = bundler;
    this.type = type;
    this.customWrapper = customWrapper;
  }

  private ParcelableCustomWrapper(Parcel in) {
    bundler = in.readParcelable(Bundler.class.getClassLoader());

    int presentValue = in.readInt();

    if (presentValue == NULL) {
      type = null;
      customWrapper = null;
      return;
    }

    type = (BundlerType) in.readParcelable(Bundler.class.getClassLoader());
    BundlerType valueType = type.typeArguments().get(0);
```

```
    @SuppressWarnings("unchecked")
    E value = (E) bundler.readFromParcel(in, valueType);

    customWrapper = new CustomWrapper<>(value);
  }

  @Override
  public void writeToParcel(Parcel dest, int flags) {
    dest.writeParcelable(bundler, flags);

    if (customWrapper == null) {
      dest.writeInt(NULL);
      return;
    }

    dest.writeInt(NOT_NULL);
    dest.writeParcelable(type, flags);
    BundlerType valueType = type.typeArguments().get(0);
    bundler.writeToParcel(dest, customWrapper.value(), valueType, flags);
  }

  @Override
  public int describeContents() {
    return 0;
  }

  @SuppressWarnings("rawtypes")
  public static final Creator<ParcelableCustomWrapper> CREATOR =
      new Creator<ParcelableCustomWrapper>() {
        @Override
        public ParcelableCustomWrapper createFromParcel(Parcel in) {
          return new ParcelableCustomWrapper(in);
        }

        @Override
        public ParcelableCustomWrapper[] newArray(int size) {
          return new ParcelableCustomWrapper[size];
        }
      };
}
```

Registering with the SDK

Once created, to use your custom parcelable wrapper you'll need to register it with the SDK.

To do this, specify `parcelableWrappers={YourParcelableWrapper.class}` in either a `CustomProfileConnector` annotation or a `CrossProfile` annotation on a class.

Future Wrappers

Future Wrappers are how the SDK adds support for futures across profiles. The SDK includes support for `ListenableFuture` by default, but for other Future types you may add support yourself.

A Future Wrapper is a class designed to wrap a specific Future type and make it available to the SDK. It follows a defined static contract and must be registered with the SDK.

In this section we will refer to the wrapped type as `T`, and the wrapper type as `W`.

Annotation

The future wrapper class must be annotated `@CustomFutureWrapper`, specifying the wrapped class as `originalType`. For example:

```
@CustomFutureWrapper(originalType=SettableFuture.class)
```

Format

Future wrappers must extend `com.google.android.enterprise.connectedapps.FutureWrapper`.

Future wrappers must have a static `W create(Bundler, BundlerType)` method which creates an instance of the wrapper. At the same time this should create an instance of the wrapped future type. This should be returned by a non-static `T getFuture()` method. The `onResult(E)` and `onException(Throwable)`

methods must be implemented to pass the result or throwable to the wrapped future.

Future wrappers must also have a static void `writeFutureResult(Bundler, BundlerType, T, FutureResultWriter<E>)` method. This should register with the passed in future for results, and when a result is given, call `resultWriter.onSuccess(value)`. If an exception is given, `resultWriter.onFailure(exception)` should be called.

Finally, future wrappers must also have a static `T<Map<Profile, E>> groupResults(Map<Profile, T<E>> results)` method which converts a map from profile to future, into a future of a map from profile to result. `CrossProfileCallbackMultiMerger` can be used to make this logic easier.

For example:

```
/** A very simple implementation of the future pattern used to test custom future
wrappers. */
public class SimpleFuture<E> {
  public static interface Consumer<E> {
    void accept(E value);
  }

  private E value;
  private Throwable thrown;
  private final CountDownLatch countDownLatch = new CountDownLatch(1);
  private Consumer<E> callback;
  private Consumer<Throwable> exceptionCallback;

  public void set(E value) {
    this.value = value;
    countDownLatch.countDown();
    if (callback != null) {
      callback.accept(value);
    }
  }

  public void setException(Throwable t) {
    this.thrown = t;
    countDownLatch.countDown();
    if (exceptionCallback != null) {
      exceptionCallback.accept(thrown);
    }
```

```
  }

  public E get() {
    try {
      countDownLatch.await();
    } catch (InterruptedException e) {
      return null;
    }
    if (thrown != null) {
      throw new RuntimeException(thrown);
    }
    return value;
  }

  public void setCallback(Consumer<E> callback, Consumer<Throwable>
exceptionCallback) {
    if (value != null) {
      callback.accept(value);
    } else if (thrown != null) {
      exceptionCallback.accept(thrown);
    } else {
      this.callback = callback;
      this.exceptionCallback = exceptionCallback;
    }
  }
}
```

```
/** Wrapper for adding support for {@link SimpleFuture} to the Connected Apps SDK.
*/
@CustomFutureWrapper(originalType = SimpleFuture.class)
public final class SimpleFutureWrapper<E> extends FutureWrapper<E> {

  private final SimpleFuture<E> future = new SimpleFuture<>();

  public static <E> SimpleFutureWrapper<E> create(Bundler bundler, BundlerType
bundlerType) {
    return new SimpleFutureWrapper<>(bundler, bundlerType);
  }

  private SimpleFutureWrapper(Bundler bundler, BundlerType bundlerType) {
    super(bundler, bundlerType);
  }

  public SimpleFuture<E> getFuture() {
```

```
    return future;
  }

  @Override
  public void onResult(E result) {
    future.set(result);
  }

  @Override
  public void onException(Throwable throwable) {
    future.setException(throwable);
  }

  public static <E> void writeFutureResult(
      SimpleFuture<E> future, FutureResultWriter<E> resultWriter) {

    future.setCallback(resultWriter::onSuccess, resultWriter::onFailure);
  }

  public static <E> SimpleFuture<Map<Profile, E>> groupResults(
      Map<Profile, SimpleFuture<E>> results) {
    SimpleFuture<Map<Profile, E>> m = new SimpleFuture<>();

    CrossProfileCallbackMultiMerger<E> merger =
        new CrossProfileCallbackMultiMerger<>(results.size(), m::set);
    for (Map.Entry<Profile, SimpleFuture<E>> result : results.entrySet()) {
      result
          .getValue()
          .setCallback(
              (value) -> merger.onResult(result.getKey(), value),
              (throwable) -> merger.missingResult(result.getKey()));
    }
    return m;
  }
}
```

Registering with the SDK

Once created, to use your custom future wrapper you'll need to register it with the SDK.

To do this, specify `futureWrappers={YourFutureWrapper.class}` in either a `CustomProfileConnector` annotation or a `CrossProfile` annotation on a class.

Direct Boot Mode

If your app supports direct boot mode, then you may need to make cross-profile calls before the profile is unlocked. By default, the SDK only allows connections when the other profile is unlocked.

To change this behaviour, if you are using a custom profile connector, you should specify `availabilityRestrictions=AvailabilityRestrictions.DIRECT_BOOT_AWARE`:

```
@GeneratedProfileConnector
@CustomProfileConnector(availabilityRestrictions=AvailabilityRestrictions.DIRECT_BOOT_AWARE)
public interface MyProfileConnector extends ProfileConnector {
  public static MyProfileConnector create(Context context) {
    return GeneratedMyProfileConnector.builder(context).build();
  }
}
```

If you are using `CrossProfileConnector`, use `.setAvailabilityRestrictions(AvailabilityRestrictions.DIRECT_BOOT_AWARE)` on the builder.

With this change, you will be informed of availability, and able to make cross profile calls, when the other profile is not unlocked. It is your responsibility to ensure your calls only access device encrypted storage.