Developing Android REST client applications

A presentation by Virgil Dobjanschi at Google I/O 2010

Transcript v0.3 by Koen Zagers, 27 December 2013 – koenzagers@hotmail.com

http://www.google.com/events/io/2010/sessions/developing-RESTful-android-apps.html

Google I/O 10 May 19 - 20, 2010
Moscone Center, San Francisco

This session will present architectural considerations for developing RESTful applications on the Android platform. It focuses on design patterns, platform integration and performance issues specific to the Android platform.

Download the PDF

Session type: 301
Attendee requirements: Advanced knowledge of Java & Android concepts
Tags: Android, Mobile, Java
Hashtag: #android6

Contents

Introduction 1
What is REST? 1
HTTP 1
Why use REST? 1
How not to do it 2
Three design patterns to do it right 3
Pattern A 4
  Bottom to top run-through 5
    The REST Method 5
    The Processor 5
    The Service 8
    The Service Helper 9
    The Activity 10
  Top-down example 11
Pattern B 12
Pattern C 14
Conclusions and advice 15
Closing words 16
Questions 16
End 16

Introduction

[00:03] My name is Virgil Dobjanschi. I'm a software engineer at Google. I work in the Android Application Group. I'm also the author, more recently, of the official Twitter app. I want to welcome you to this session. Today we're going to talk about the development of REST client applications. Before we begin, I would like to invite you to view live notes and ask questions regarding this session by using Google Wave. I want to give you just a few seconds to write down this URL if you haven't done this already. While you're doing that, I just want to gauge your opinion about the Froyo announcement. Were you guys impressed by this announcement, by everything we announced today in the keynote? (Applause) Thank you. Well, I work in that group and I'm impressed. I'm amazed by the amount of stuff that happens in that group, and I invite you to go to all the other Android sessions this afternoon, learn about Android Cloud to Device Messaging, and all the other great lessons you can learn and you can use when you're developing Android applications. We have a lot of content to cover, so let's get started.

What is REST?

[01:26] So what is REST? REST stands for Representational State Transfer. It is a broadly adopted architecture style. This architecture style is comprised of clients and servers. Clients make request to the server to get or change the state of a particular resource. Servers respond to the client with representations of resources. So what is a resource? A resource is any meaningful concept that can be addressed. For the purpose of this talk I'm going to reference back to the Google Finance API as a simple example of an API. A resource, in the case of the Google Finance API, is a stock portfolio, which is characterized by its name and the currency in which it operates. The representation of a resource is simply a document that fully describes that particular resource. In case of the Google Finance API, this is an XML document which simply describes that particular resource in terms of its name and the currency, and maybe its ID for the purpose of referring to that particular resource later on, when you need to update or delete it.

HTTP

[02:45] REST is in no way associated with HTTP. You can use any transport protocol, but certainly HTTP is the most commonly used transport protocol for the REST style architectures. There are a large number of REST API's available today. Google Code offers already a host of such API's. Social networking sites, such as Twitter, Facebook, MySpace, already have a very rich REST API. You can find on the web all kinds of information based REST services for calendars, flight schedules and so forth. The first question you're going to ask yourself before you start to development of a REST client application is, why would I possibly develop an application like that if the service that I'm connecting to already has a mobile friendly website? In other words, why don't just use the browser to connect to that service, and I'm done?

Why use REST?

[03:55] Well, until browser technology catches up, I have five reasons for you. The first one is that your application, your Android application, will be able to integrate with the Android platform. It will be able to use intents, content providers. It's going to be able to access all the private API's available today only to Android applications. Things that, at least at this point, you can't do from a browser. Your application will be able to invoke intents. Not only to pick photos, but many of the other intents that are available today for all applications to use. The second reason is, your application can offer intents to other applications. In other words, you're going to be able to enrich the behavior of the

platform by offering new functionality to other applications. Content providers are there for you to access. The contacts Content Provider is a very popular Content Provider. Lots of applications use it. The third reason is, your applications can run in the background. What does that mean? It means that if you would like to have your application refresh the data from the server and new data is actually retrieved from the server, your application has the option to present a notification to the user to let them know that that particular data is available. A browser can't do that. Your application runs on devices with limited connectivity. What does that mean? You're going to try to initiate some of the REST methods and they will sometimes fail, because you operate in environments like here in the conference where the network comes and goes. An average application has the option to run in the background and retry operations in the background by using an alarm for example, for the purpose of relieving the user from trying to hit that refresh or post button in the browser. Your application can actually be faster than a web browser. Your application has the option of retrieving all that network content in JSON, or some binary, or some XML format, parse it, and store it in a database. From there on, when you want to retrieve new content, you may have the option of retrieving only content that's newer than the one you already have or older than the one you already have, but not the same data. You're also not retrieving all the html that comes along, or any kind of javascript that might be quite big and take too long to download. Finally, your application has the option of being consistent in terms of user interface with the rest of the operating system. You can innovate in many ways to deliver a fantastic user experience to the user. If you have participated yesterday in the user experience talk, please take advantage of those UI patterns. Your application is going to be so much faster and easier in some cases to use than the browser. And finally, when users are presented with the option of using a web browser versus a REST application, I can guarantee that most users will choose the REST application.

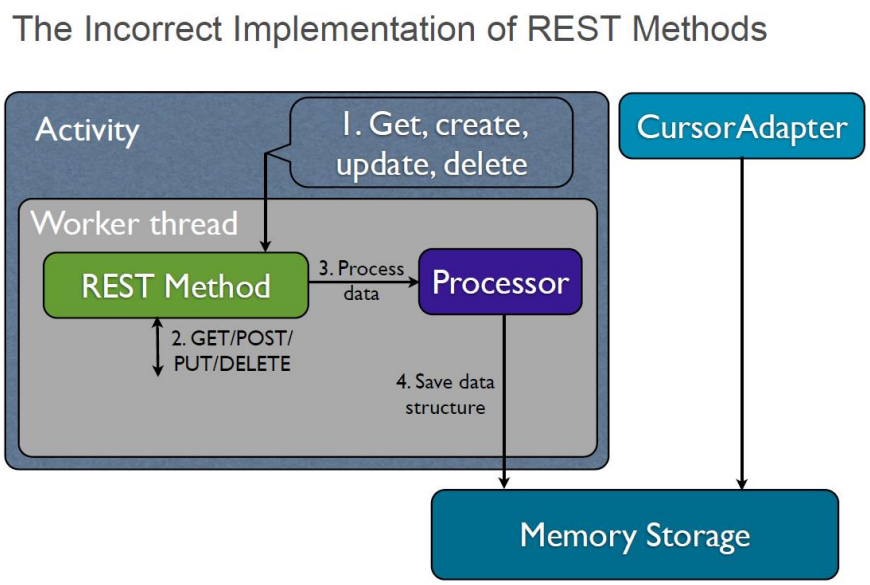
How not to do it

[07:37] Before we begin I would like to describe the way a novice programmer would approach the development of a REST application. That's not you. Just for the purpose of this discussion, let's look at these things. Hopefully it won't take you ten-thousand tries to get there. But I can tell you that I make these mistakes myself. It's actually human nature to try to learn as little as you can about a particular thing and move quickly forward and say "OK, I'm going to have this app ready in ten minutes", and I'm going to say: slow down. Get all the facts first. Learn what's correct. Implement it correctly so you have a good base for your application to get going.

[08:28] So you're a Java programmer. You have looked around in the Android SDK a little bit, and you say to yourself "I know what an activity is, how hard would it be? I'm just going to create this activity, I understand a REST method takes a long time to execute because I need to connect to a server so I'm going to run it in the context of a thread." You decide that this thread is going to be an inner class of your activity. And you decide that you're going to store all this data that you retrieve from the server in memory. Maybe the application I'm describing here is your attempt at displaying the list of stock portfolios that I have already set up for my own account in Google Finance.

So you get your application to run, it runs fast, it doesn't crash, so why is this not the correct approach? [09:22] You have to understand how the Android operating system works. The Android operating system was designed to work on devices with limited memory. What does that mean? It means that for example when the operating system is attempting to start a new application and it ran out of memory, it needs to make a decision about forcefully shutting down an existing application. So how does it make that decision? The answer is: you have to help the operating system make that decision. If your application does not have a foreground activity running, in other words a visible activity running, or a service running, the operating system can say OK, this application is not

currently displayed to the user, it's not currently performing any operations, so it's safe to shut it down.



[10:22] So, let's now go back to our diagram and try to understand what's wrong with this picture here. Here's what happens here. You launch the thread that initiates the REST method, and guess what. You pause your activity because maybe an incoming call needs to be answered. What happens is, the operating system says oh, this particular process doesn't have any foreground activity so you know what, I'm going to kill it. I'm going to forcefully shut it down. Well guess what, your POST, PUT, DELETE method maybe has executed on the server and you'll never learn the result of that particular method in your app. You execute your GET method, get all this data in, you parse it, you're happy. The operating system will shut it down for you. Wasted bandwidth. Another thing that's wrong with your approach is you decided to store the data in memory. So you're saying well hang on, my application is fast. The reason why that's not a good idea, is because by having to repeatedly retrieve data from the server, either because the user has simply restarted the device, or simply because at one point your process had to be forcefully shut down, you're wasting CPU, you're wasting battery, you're wasting network bandwidth. That's not a great way to write an Android application. And for those of you who are going to say well, it's much faster to get this data from memory, I'm going to say, by using the Content Provider you have the option, should you consider that necessary, to cache this data in memory. So that's really not a good reason to say, I'm not going to store my data. As I said, none of you will probably write the application this way.

Three design patterns to do it right

[12:25] Let's describe today three design patterns that you can use in your applications that would give the best user experience and the best performance. These are not the only three patterns you can use. Once you look at them, once you understand what they are all about, you can possibly create your own. As long as your application complies with the basic principles that we're laying out here for you, it's OK to continue to innovate from here. We're not going to talk about implementation details, I'm not going to show you code. Hopefully you like that.

[13:11] So let's introduce these three patterns. One is based on the Service API, one is based on the Content Provider API, and the third one is simply a variant of the second where we're going to use a Sync Adapter.

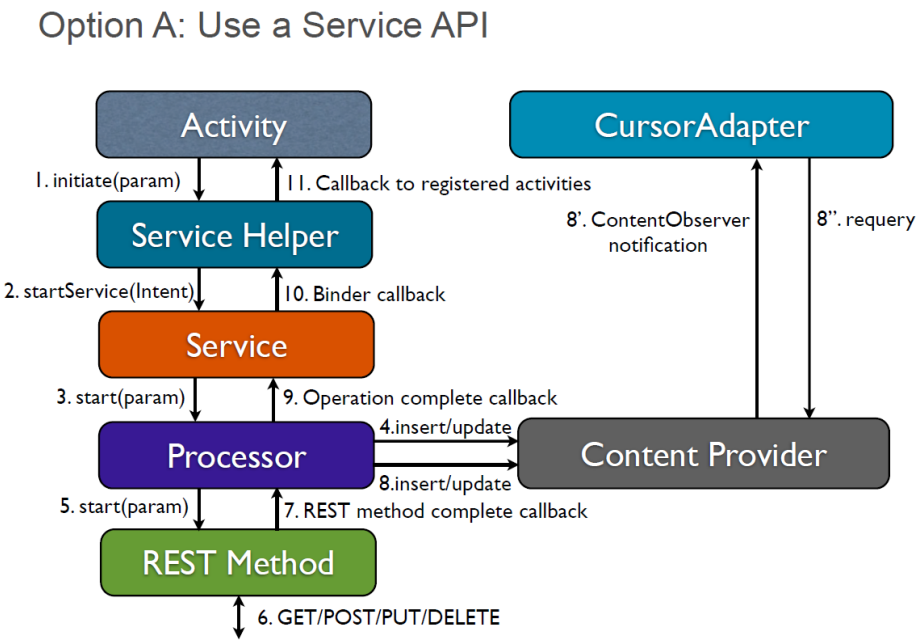
REST Method Implementation Patterns

- Introducing three design patterns to handle REST methods
  - Use a Service API
  - Use the ContentProvider API
  - Use the ContentProvider API and a SyncAdapter

This is an advanced talk. Last year we got a lot of feedback from you guys saying that some of the sessions were too light. We decided to compensate this year. Even though it's an advanced talk, I'm going to try to introduce these concepts which some of you may not be familiar with, so you understand the basic idea behind it.

Pattern A
Implementing REST Methods
Option A: Use a Service API

[13:47] So let's get started with the first pattern. The first pattern makes use of a service. Again, for those of you who are not familiar with a service, a service is an Android component that does not interact with the user and is specifically designed to execute operations in the background. It runs in the context of the main thread, we can call that the UI thread. And in order for you to execute these operations you need to still create a worker thread in the service. Some people think that by simply starting the service, I can simply execute in the context of the thread of the service that particular long running operation. That's false.



[14:35] So now that we understand that the service exists in the platform, and has by the way a very simple API, you can either implement the service's API by using an Intent API or by using a Binder interface. For the purpose of this talk, we're going to use the Intent API, which is by the way a little bit easier to understand.

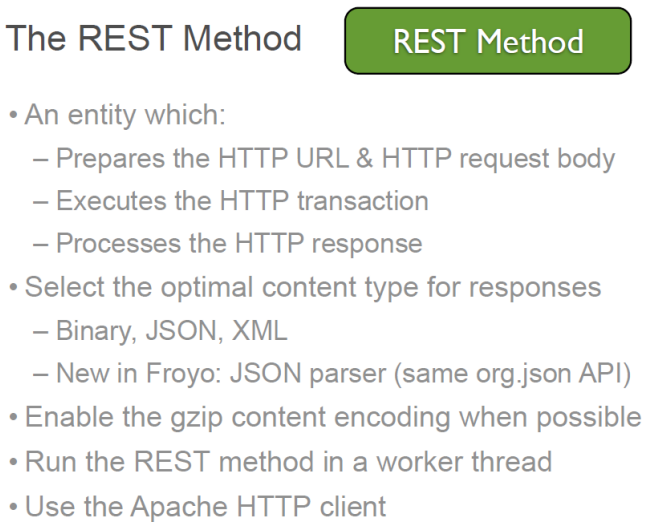
Intent API or Binder interface?

We can now get started with the first pattern. This diagram is not exactly simple. To best understand it we're going take a bottom-to-top approach by explaining the role of each component, and once we're done we're going to do one example starting from the top, running through an entire call flow to see how this works.

Bottom to top run-through

The REST Method

[15:20] So let's get started with the REST Method. What is the REST Method? Really simple. It's an entity which has the ability to prepare an HTTP URL, to prepare the HTTP body in some cases, execute the HTTP transaction with the server, and process the response which typically means parse the response from the server. There's not much to say about this.



Let's make some performance remarks. Many REST API's offer today the ability that you select the content type of responses. Prefer them in this order: some binary format - AMF3, whatever is out there -, JSON, and XML. We're happy to announce that if you choose JSON we have a brand new implementation of the org.json API. It has exactly the same API as before but it performs much faster; there is very little garbage collection going on. For those of you who have tried the older version, you're going to be very happy with this particular implementation. Enable gzip if the REST API that you're using allows it. Gzip is a very fast library, it's a native implementation, sometimes you get compression ratios up to 5:1, 10:1, depending on how much data you're retrieving. And by doing so, you're simply speeding up the download of data that you want plus the battery life will be preserved better because the radio will be used for a less amount of time. Always think about the radio, always think about the network and the implication it has on the battery life of that particular device. Always run this REST method in a worker thread. You understand that an HTTP transaction takes time to execute, sometimes a long time if a timeout occurs, and you should never run such operations in the context of the main thread. And finally, use the Apache HTTP client, not the Java URL connection. So what do we have so far, we have the REST Method, a simple API, it fires back a Java listener callback, really simple.

Enable gzip

Always run a REST method in a worker thread

Use Apache HTTP

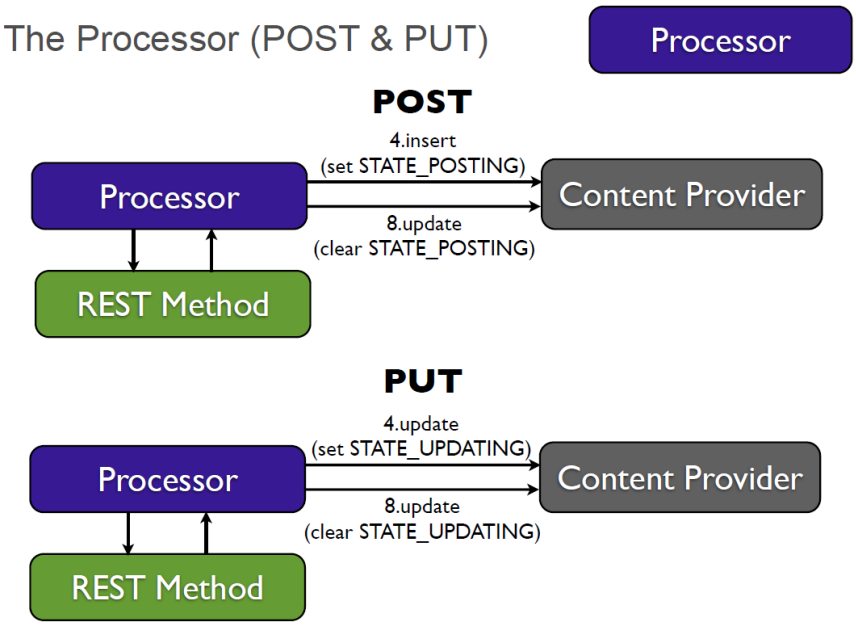
The Processor

[17:41] Let's take a look at the Processor. The role of the Processor is to mirror the state of server resources in your local database. We're going to use a database. The role of the Processor is to say, OK, this is the state of this particular resource. To better understand how the Processor works we need to create the concept of a resource in the database. Well, it's really simple. It's exactly one row in the database. In the case of Google Finance it's a row that maybe has the ID of this particular resource, the name of this particular portfolio, and the currency. We're going to add two more columns to that: one, a status column, and two, a result column. The status column is the one that simply indicates the transitional / at rest (or steady-state) of that particular resource. What that means is that every resource when it executes a REST method has a transitional state. While you are

executing this particular transaction the resource is not yet in sync with the server, so we want to keep track of this particular state. Why would I possibly need this? Two reasons. Your user interface may want to display maybe a little icon next to the resource. I don't have to stay in the activity that actually started this POST for creation of the portfolio, I could just leave that activity. And my list of items is just going to display a little icon next to that particular resource saying "I'm in the process of synchronizing the resource; but go ahead and use the app, I'm going to let you know when that's done". When that disappears, that's a good indication that your resource has now been synchronized.

State icons

So in the status we're going to hold some flags. These flags are going to say "I'm in the process of executing a POST method for this particular resource", "I'm in the process of updating", "deleting" this particular resource. And we're going to use one more flag to say we're in the process of executing a HTTP transaction for this particular resource. Now you have a full description of what's going on. At any time you can just look at the database and say OK, I know exactly what's going on with this particular resource. The result column can simply hold the HTTP result of the last REST method you executed in regard to that particular resource.



[20:15] Let's come back to the Processor. The Processor executes before and after the REST Method executes.

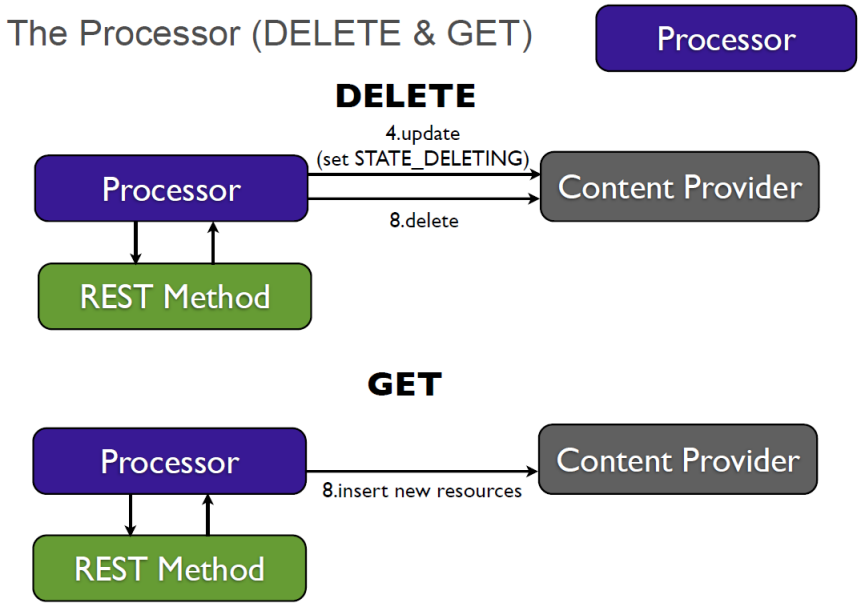
Before the REST Method executes, in case of the POST method, it's simply going to create a row in the database by calling an insert method, to create the row that corresponds to the resource we need to update. We're going to set the flags to POSTing, meaning we're in the process of executing a POST method for this particular resource. We're going to say we're also transacting by the way, so we're going to set that flag too in the status column, and we're finally going to initiate the REST method. The REST method completes, let's say successfully. We're going to update the row in the database with the help of the Content Provider. We're going to clear the state POSTing flag because that particular method is no longer pending. And lastly we're also going to clear the transacting flag so we know we're no longer transacting with the server.

PUT works very similar. It's hardly worth going through this, but before the REST method executes we're going to update this particular row in the database. We're going to set the updating flag meaning we're in the process of updating this resource on the server. Set the transaction flag, and

execute the REST method. Once it is done update it again and clear the two flags that we just set if everything goes right.

By the way, as you all know most correct REST API implementations are idempotent. This means that when you send a document to the server saying "I want to change the state of this resource on the server" and you send the change you want to make, the response that comes back is the full description of that resource as the server sees it. In other words, the state on the server side. That's why it's recommended that you go ahead and update again, although many times that's not necessary. It's up to you. But either way in this case you're going to have to clear the flags.

Idempotence



Delete, very simple. Set a flag to that particular resource for deletion. You're not actually going to delete the resource yet. Why? Because we have not yet successfully executed the REST method. You go ahead and execute the REST method, it succeeds and finally you delete that row. You're done. Your user interface by the way has the option of not even displaying that particular resource anymore. Or you have the option to display a little synchronizing resource icon next to that resource to indicate that it's going to be deleted.

By the way, these flags can serve a different purpose. What happens when you need to retry these operations? Wouldn't it be great to know what exactly you need to do? The posting flag would mean you need to execute a POST message. Look at the transaction flag, is there a HTTP transaction already POSTing? If that's the case, I'm not going to start another one. Otherwise, I have the option of saying let me retry this, maybe the network didn't work the last time. But it's a great way to mirror the state of the resource in the database and retry these methods.

GET is the simplest one of all. The Processor has nothing to do before the REST method, inserts the new resources once it gets them from the server. This could be the list of stock portfolios. Maybe you have many of them on Google Finance.

[23:55] One more thing to say about processors. Never execute database operations or Content Provider operations in the context of the main thread. Sooner or later you're going to run into application not responding errors, the so-called ANR's. It's not good practice. Learn to stay away from that.

Never execute database operations or content provider operations on the main thread

Also, use transactions when you use Sqlite. Not only will they preserve data integrity, but they will increase the performance of your database operations.

Use database transactions

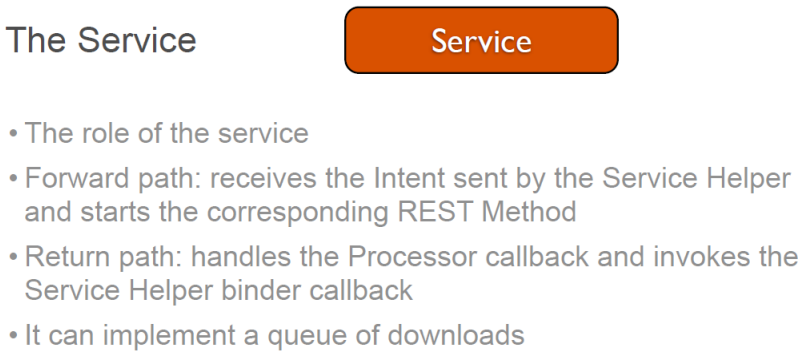
We have the Processor, we have the REST Method, the Processor has almost the same API as the REST Method it goes through, but this time we're able to mirror the state of that particular resource in the database.

The Service

[24:47] We said at the beginning that it's not a good idea to start long running operations in the activity. That's where the Service comes in. The role of the Service is to simply allow your application to execute operations in the background while an activity is long gone. I'm going to say it again, an activity comes and goes, it's just the user interface of your application. The user has the option to hit the Home button, hit the destroy button, in other words, go back home. Your long running operation is still executing, and the nice thing is that you have the database simply store the state of that particular resource, and when you come back to your application, regardless of what happened, you can simply learn what happened. Did it work? Is it still pending? What's going on?

On the forward path the Service simply receives an intent – we said we're going to use the Intent-based API – it unpacks the content of that intent; just think of it as a map. I passed in for example the name of this particular stock portfolio and the currency, I'm going to extract it from there, make the Java call to the Processor, done. On the return path, I'm just simply going to handle the Processor callback, and I'm going to invoke what is called a Binder-callback. What's a Binder-callback? Think of it as an interface that was passed in the request intent. So the upper layer says, when you're done with this operation I want you to invoke this Binder-callback and let me know that everything worked or failed.

Binder-callback



Here's another thing you can do in the Service. You write a social application, and suddenly your UI needs to download ten, twenty images because you're scrolling really fast through that list. Is it a good idea to download ten, twenty little images at the same time? The answer is no. Be nice to all the other applications, they might need the bandwidth at that time. Implement a little queue in this particular service. Allow maybe one, two, or three parallel downloads of images, especially if they are larger. Always think of the other apps when you are writing one.

Queue downloads

And finally the Service is a stateless component. Everything it needs, it receives in the intent. It fires the callback. Unless you implement the queue we were just talking about for download, there's no reason to keep any state.

And most importantly, when all the pending operations have completed, shut down the service. If you don't, the operating system is going to say well, this guy is busy, let him keep the memory that he wants, let him do whatever he needs to do. Be nice again, to all the other applications. Shut down

Shut down the servcie

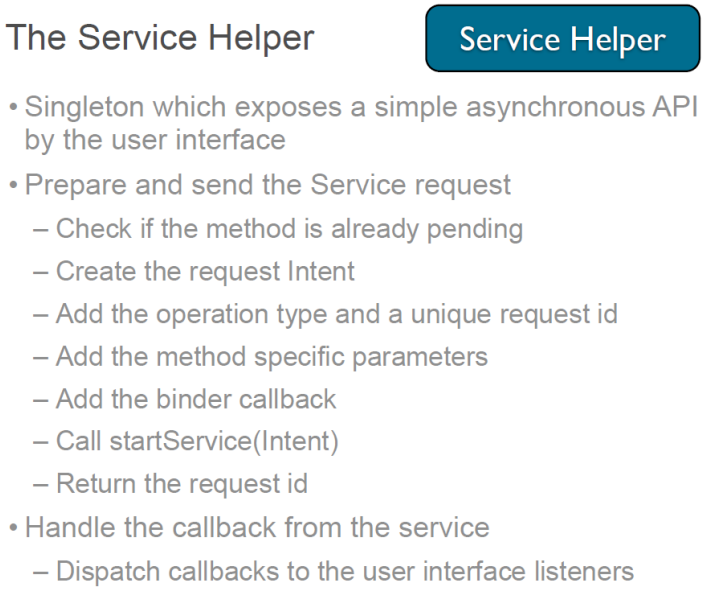
the service once you're done. It's very easy for the Service Helper to stop the service, or the service can stop itself. There are many ways to do it, you have no excuse to leave that service running.

Almost there.

We have a Service with an Intent API and it asynchronously fires a Binder-callback interface. Sometimes preparing these intents is not that easy or at least not elegant, I know that. And handling the Binder-callback is not something we should leave as an exercise for the reader. What we should do is have a very thin layer we call the Service Helper that prepares these intents for us on the forward path and handles the Binder-callback on the return path.

The Service Helper

[28:36] The role of the Service Helper is to expose a very simple asynchronous API to the caller. Let's say we want to create a portfolio. The name of the method is createPortfolio; here's the name, here's the currency. It returns the request ID which uniquely identifies that particular pending REST method. It's a singleton, there's nothing to it. You make this call, and here's what it does. It's much simpler than it looks.



Check for pending operations

The first thing you may want to do is have the ability to ask the question: is this particular REST method already pending? It's certainly not the case with create, but if you do maybe get my stock portfolio list, what if you issue this request, and some other activity which quickly you switch to issues another request, well you certainly don't want this particular situation where let's say you're downloading and parsing the stock portfolios in parallel in two different threads, it makes no sense. You can protect yourself in this particular pattern here in the Service Helper. You can implement a map from the request ID to this intent, and you can always first of all ask the Service: is this particular request ID still pending? You can do something else. You can look at the values, the intents in the values method, and say by looking at the intents you will be able to answer the question: is this particular type of operation, maybe even with these parameters, already pending? And if yes, you'll simply return that request ID to the user. It's a nice way to get around some of these problems.

And finally you put operation type, you put any method-specific parameters such as name and currency, you set the Binder-callback and you finally call startService which simply sends that intent to the Service that we discussed earlier. On the return path you get the Binder-callback and you say, this particular operation is no longer pending, and you can dispatch a callback to any listeners.

Listeners are typically going to be activities that would like to know the result of this particular REST method.

One interesting discussion here is what kind of data should I pass from the Service to the Service Helper. And you're going to say OK, you just retrieved that list of stock portfolios. Should I just pass it on? Should I just move all that data through this particular Binder-callback? Well, think of it as a marshalling interface. In other words, something that has to cross process boundaries. You don't want to do that. Keep as little data as possible in that particular callback. For example, you can say, the result of this particular REST method was 200 OK, you can say here's by the way the original intent that I used to start this operation, maybe the caller would like to analyze the content of that intent to remind itself what it was executing at that time. All the data you already retrieved is already in the database. The activity has the option of simply saying OK, thanks for letting me know, I'm going to go and retrieve that content. There's absolutely no harm done if you choose once in a while to pass some data through this Binder interface. You can use any parcelable - think of it as a serialization technique - to send data over this interface. The less data you parcel the better it is.

We're almost done. The Service Helper has a simple API. If there are any listeners that registered, these callbacks go back to the activity. Let's look at the activity.

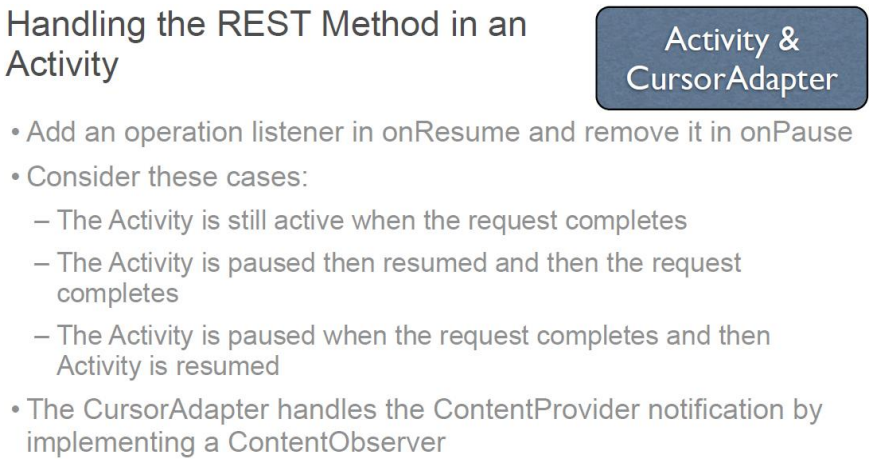
The Activity

[32:38] Always remember the activities are paused, resumed, or destroyed. They have their own mind. They have, better said, their own life-cycle. Think of them, again, as just user interface that is controlled by the user, and in some cases activities pop in front of your application just because maybe a phone call comes in in which case the application will be paused. In onResume we have the option of adding a callback to the service to say OK I'm back, I'm interested in what you have to say. onPause remove that, because if a callback occurs while the activity is paused, sooner or later you're going to get a crash. You're trying to access this activity while it's paused and that's certainly an incorrect technique.

Remove callbacks onPause

Always consider these three cases:

A request method is started by the activity. The activity just sits there in the foreground, everything is nice, it gets the callback from the Service Helper, easy. In case of an error you can say, I'm sorry I couldn't perform your operation, or hey, cool, we created the stock portfolio. What if the activity starts the REST method, pauses, comes back, resumes, and now the callback occurs. Well, that's not much more difficult. You can still handle the callback, everything is great. In this case you may need to store the request ID so you know to ask the Service Helper "is this particular request still pending?" And so in onResume the answer this time is going to be "yes, that's still pending". OK, so we'll just wait for the callback. Nothing special.



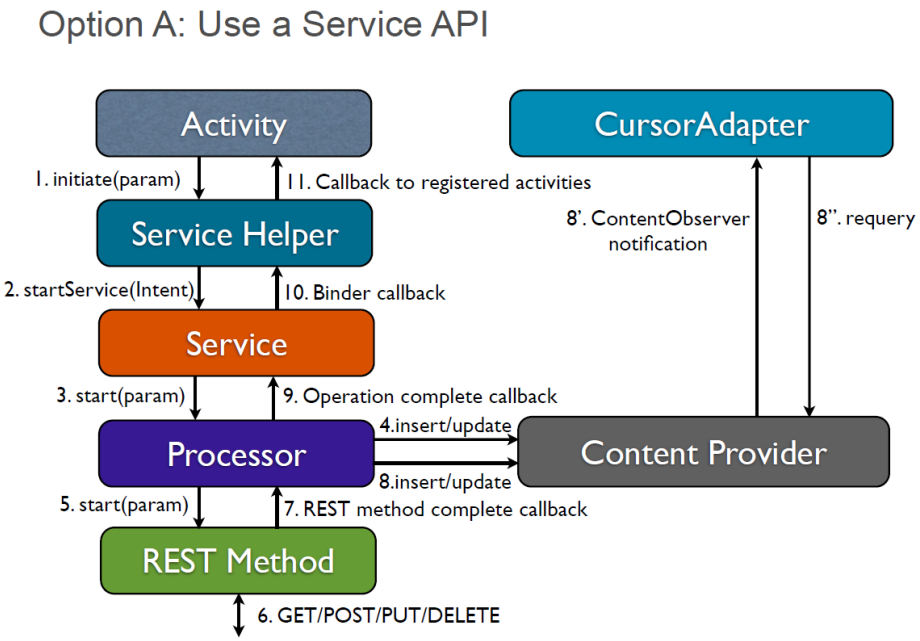
The third one is the most complex. REST method is just started, the activity is paused, REST method completes while the activity is paused. The activity comes back to life and resumes. What's happening? Well, in onResume this time you're going to have again the request ID that you started and you're going to say "is this request still running?" And this time it's going to be no, it's not. So now you understand why we chose to store the result in the database. The application has the option to go back to the database and figure out exactly what happened with this particular pending request. While it was maybe destroyed, maybe paused, and so on. Remember, that long running operation - the REST method - was running because it executed in the context of a service and the operating system said OK I'm good, I'm going to leave this process alone because it's executing what it thinks is important. So that's great.

On the return path here we have the option to receive callbacks from the Content Provider itself. The Content Provider supports what's called ContentObservers. ContentObservers are simple ways to receive notifications from the Content Provider when data changes for a particular set of entries in a particular table. In other words you can monitor the status of a particular row, or maybe all the rows, all the resources in that database.

Content Observers

Top-down example

[36:06] We reached the top. Let's take a full example and run with it.

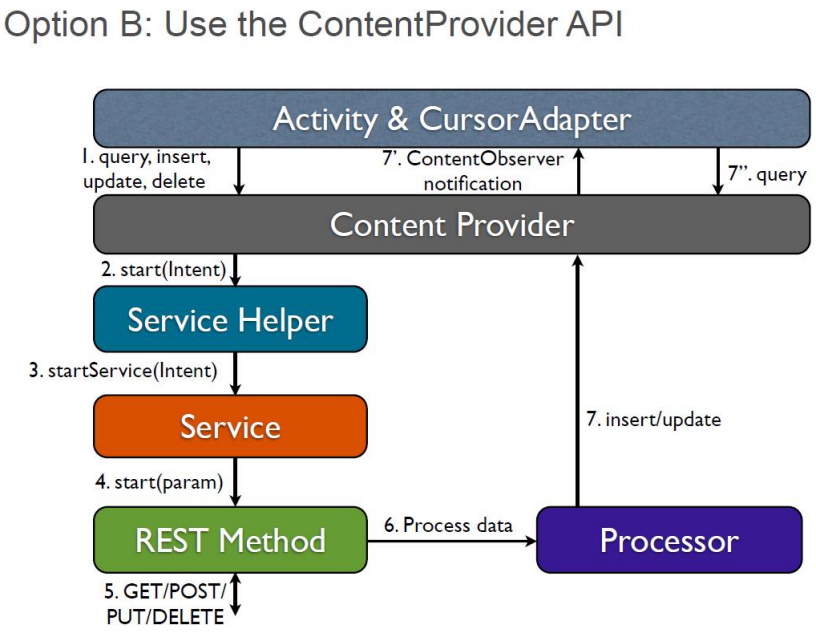


The activity decides to create a portfolio. It says OK, I'm going to call the simple method in the Service Helper called createPortfolio. The Service Helper says OK, I'm going to new an intent, pack in it everything I need, the name of the service, the currency, the Binder-callback, the type of operation, I'm creating a portfolio, the unique request ID, it calls startService, it says this particular operation is pending, and returns the request ID to the caller. The caller has the option to use this particular request ID to find out if this particular REST method is still pending. Finally the Service gets the intent, the Service unpacks it, makes the Java call to the Processor. The Processor says, what kind of method corresponds to this particular Java method? Ah, this is a POST, I need to create a new stock portfolio, so I'm going to go and insert that content into the database, in other words I'm going to create a row in the database, I'm going to set the status POSTing, set the transaction flag in the status column, and I'm going to say OK, I'm ready to execute my REST method. The REST Method prepares the URL parameters, prepares the request body if that's required – and for Google Finance that's certainly the case – gets the callback once the REST method transacts with the server, it obviously gets the parsed result back from the REST Method, the Processor says OK, I'm going to update the state of this resource in the database, so it updates the content, it updates the flag by saying OK this particular method is no longer pending, we assume obviously that it worked, and the Content Provider would fire notifications as needed if there are any kind of ContentObservers. On the return path the Processor says to the Service "thank you, I'm done." The Service says "OK, I'm going to invoke the Binder-callback that was already in the intent" and finally the Service Helper says "you know what, this REST method is no longer pending, I'm going to let any activities know about the result." And the activities have the option to handle that callback and display an error to the user and so forth. So that's a quick run through this particular design pattern. Next one.

Pattern B
Implementing REST Methods
Option B: Use the ContentProvider API

[38:26] Based on a Content Provider API. What is a Content Provider? For those of you who don't know, it's a simple way in Android to store and retrieve data across process boundaries. Typically a Content Provider sits in front of the database. It offers the insert, update, delete and query methods that in the end access the database. Really simple stuff. I can't go into details, hopefully this explanation will be sufficient for the purpose of this discussion.

So the first question you're going to have for me: OK, how in the world did you get to the conclusion that you can use a Content Provider API to initiate REST methods? Well, the answer is simple. There is a one-to-one mapping between REST method-types and some of the methods in the Content Provider. The GET REST method corresponds to query in the database, the POST REST method corresponds to insert, the PUT REST method corresponds to update, and finally the DELETE REST method corresponds to delete. Really simple. It's not hard to imagine how this works.



So here's the pattern that we can use. By the way in this particular diagram the Service Helper, the Service, the REST Method, work just as before. The Content Provider and the Processor have a slightly different functionality that we can cover once we go through an entire flow for this diagram.

[39:55] So let's assume as before that we would like to create a stock portfolio. The activity says OK, this is going to be a creation so that corresponds to a POST, so obviously I'm going to do an insert. It prepares the content values - for those of you who are not familiar, it's just a way to specify the values that you would like to insert in specific columns - in the Content Provider, and calls insert. The Content Provider says OK, I'm going to create this row for you. Before it returns however, it does a trick and says I understand that this particular operation needs to execute a POST REST method, so it initiates – with the help of the Service Helper – the POST method. It returns the URI of the newly inserted row to the caller – the activity in this case – and it simply says OK, I inserted it for you and by the way, I'm going to keep the transactional state for this particular resource, the state POSTing will be on, the transacting flag will be on, and finally the activity has the option to use these flags, should it need, to display the status of this particular resource while it's transitioning in the UI. That's up to the application. Or choose not to show it at all, or whatever policy it would like to institute.

Finally the Service Helper calls the Service, executes the REST method, once the REST method successfully executes the Processor says OK, because the REST method succeeded I'm going to clear these flags, I'm going to update the row, and the activity - maybe a CursorAdapter, any ContentObserver – will have the ability to understand that the status of this particular resource has been updated.

[41:41] By the way, in this case, if the REST method fails, it's the responsibility of the Content Provider to say OK, I'm going to set up an alarm here and maybe after five minutes I'm going to retry it. And maybe I'll implement some nice exponential back-off so I don't keep retrying and kill the battery in the process of doing so.

Please note that in this particular pattern we broke the Content Provider contract a little bit. For insert and update everything works as expected. You're calling insert, we insert the resource right away into the database as a row. For update we do the same thing, we update the resource at the database. For query (get) we do something funny. We go to the database first, we get the rows that already exist, now we have a cursor – and this is the cursor that a query typically returns to the caller

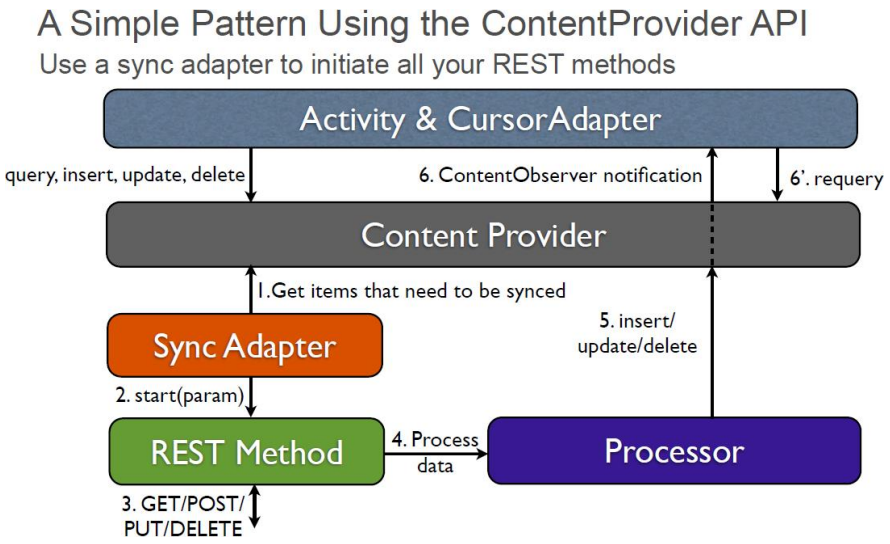
– now, if the activity would like to retrieve brand new fresh content from the server he has the option in the query URI to set a flag that says I would like to also get the latest data from the server. So the Content Provider says OK, not only am I going to give you the results in the cursor, but also I'm going to let the Service Helper know that it's time to execute a GET method, maybe in the refresh mode, to say just give me items that are newer than the ones I have. Delete breaks the contract. For delete you say I'm going to tag this row for deletion, I'm not going to delete it right away. And finally the row gets deleted once the particular DELETE REST method has completed. It's not a big deal, you just need to think about it. And maybe once you display your results in a list activity with a cursor, again look at the flag to see what the state is of that particular resource in the database.
There are other ways to do this. You can remove the row from the database, possibly move it to another table where you're going to be able to retry delete requests. Again, we're not forcing you to adopt these particular design patterns.

Pattern C
A Simple Pattern for the REST of us
Option C: Use a ContentProvider API and a SyncAdapter

[43:56] The last pattern is simply a variant of the previous one. We're still going to use the Content Provider API, but we're going to use the help of a Sync Adapter.

The Sync Adapter is a concept that you should learn as soon as you get home and you start developing for applications. What it is, is the ability of the system to help you synchronize remote and local content. The system employs the help of a Sync Manager. The Sync Manager manages all the Sync Adapters from across all apps. The Sync Adapters are simply services started by the Sync Manager. The Sync Manager implements a queue of Sync Adapters. In other words if you say I would like to sync my data, there's a requestSync method in the Sync Manager. It does not mean that the Sync Manager will say sure, right away. What it does instead, it queues the execution of this particular Sync Adapter behind any other Sync Adapters that need to execute.

By the way, even if the queue is empty the Sync Manager has the option of saying OK, I'll get to it. Why does it do that? Why doesn't the Sync Manager just run out there and start your sync operation right away? The reason is, it's trying to preserve the integrity of the entire system. What if you start your Gmail sync with the e-mail sync, you develop your own app, all of a sudden you have four apps executing the sync operations in parallel. They all do network transactions, they all do parsing, they all do database operations, it's a mess. Don't do that. Obviously the Sync Adapter is going to help you, it's going to prevent you from getting into that trouble. But the characteristic of a Sync Adapter is that it does not execute maybe as fast as you'd like to. So with that in mind, let's look at this pattern.



[46:15] The activity works the same way. You need to create a stock portfolio, you call insert into the Content Provider, the Content Provider creates that row in the database, sets its transitional state to whatever it needs to be, that's it. It returns to say thank you, I'm done. So how does the REST method execute? That's where the Sync Adapter comes in. Well, I lied. The Content Provider, before returning, calls requestSync. It says when you have a chance, come back to me, start my Sync Adapter. The Sync Adapter starts, looks in the database and asks, are there any resources that are in the transitional state at this point? Yes, here's our resource, it needs to be POSTed. It reads the row from the database, starts the REST method, executes it, gets the result, updates the database, and says done.

What if it fails? Well, awesome. The Sync Manager has the ability to handle errors nicely. The Sync Adapter is responsible for simply throwing errors back at the Sync Manager. The Sync Manager says, there was an error while I tried to sync this, you know what, by implementing exponential back-off I'll queue this particular Sync Adapter behind any others, if any, and I'm going to restart the sync when I have a chance. All that is already there for you. You don't have to write your code with alarms, you don't have to do all this stuff. All you have to do is learn how to report errors from the Sync Adapter to the Sync Manager.

All our applications use the concept of the Sync Adapter to refresh content. Gmail, e-mail, all these apps use that particular concept. Please use it in your apps.

Conclusions and advice

[48:06] Conclusions slide.

Conclusions

- Do not implement REST methods inside Activities
- Start long running operations from a Service
- Persist early & persist often
- Minimize the network usage
- Use a sync adapter to execute background operations which are not time critical
  - New in Froyo: Android Cloud to Device Messaging

[49:07] Don't let your database grow infinitely; purge old data; don't hold a lot of data in cursors

[50:15] Use paging

Closing words

[51:20] Closing words

[51:42] Thank you

Questions

[51:50] Questions

[52:41] First question

[55:14] Second question - IntentService class

[55:40] Answer

[56:14] Second question (2) - GET

[56:40] Answer

[57:22] Wave users' questions

[57:36] Wave question - Apache HTTP client vs Java

[57:44] Use Apache HTTP because more robust

[58:12] Third question - search

End

[59:29] End