Networking

Joe Rogers
Android 310

Networking

- One of the most common tasks an app performs to provide data to the user.
- Usually done via HTTP(S) protocol, but other techniques are possible.
- It is also one operation that poses more challenges in order to balance user experience, battery life and bandwidth.

The hidden stack switch

- Until Android 2.2, the best HTTP stack was the AndroidHTTPClient based on Apache.
  - Great for devs familiar with Apache Http
  - In 2011 with Android 2.3, a blog post announced Apache would no longer be maintained and all apps running on 2.3+ should use HttpURLConnection.
  - However, 2.2 support still required apache http.
  - In March 2015, Apache was officially deprecated with Android 5.1.

HttpURLConnection

- Only stack maintained or updated since Android 2.3 (2011)
    - 2.3 added transparent gzip support
    - 4.0 officially added the HttpResponseCache (existed since 2.3 but had to use reflection to create)
    - 4.4 Completely redid the internal plumbing to add SPDY(now HTTP/2) support.
    - As of 5.1 only supported stack built into Android.

Permissions

- There are two basic permissions you need to perform networking.
  - android.permission.INTERNET
    - Provides access to internet. (If forget, you will see an exception that asks if you forgot).
  - android.permission.ACCESS_NETWORK_STATE
    - Optional but recommended
    - Needed to determine if you have a valid connection, or type of connection (wifi, etc)

Caching

HttpResponseCache

- Officially added in Android 4.0
- Fully compliant with HTTP RFC headers, but will not cache partial responses.
- Not configured in app by default, requiring the app to create before use.
- App also responsible for “flushing” cache.

Cache effectiveness

- Server should send one of the following in this order:
  - Cache-Control with a max-age other than 0
  - Expires with a date in the future. (HTTP 1.0)
  - Last-Modified. This still causes a server call as the cache will issue an If-Modified-Since request to see if the cached data is still valid.
- No obvious caching may require the app to manually cache to avoid excessive calls.

Security

HTTPS first

- As a developer, should use HTTPS whenever possible.
  - Prevents data from being sent in the clear (including api keys)
  - Requests containing location, personal information, etc should absolutely be encrypted via HTTPS.
  - This is a common mistake. Perhaps because initial dev servers may not support HTTPS and devs forget.

Updating the security provider

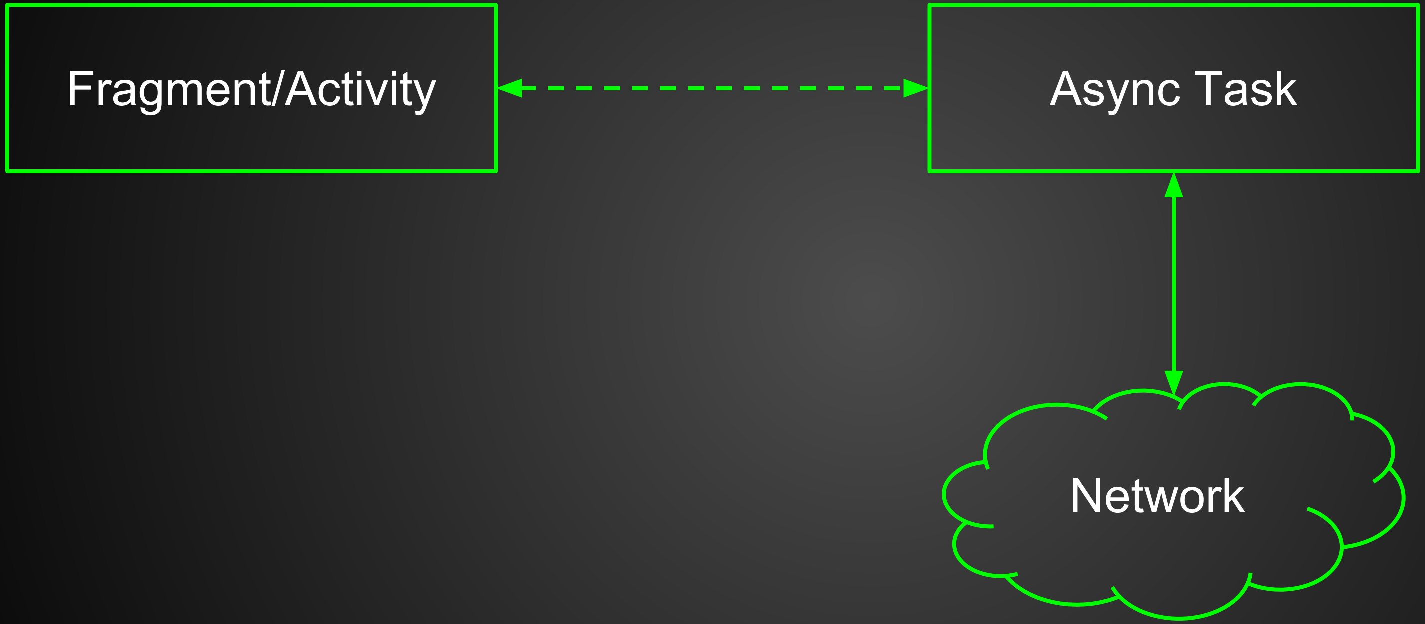
- As part of Google Play Services apps can now update the security provider to ensure the latest updates are installed.
- Perform before making any network calls.
- If the call “Fails” you should assume that the provider is possibly out of date and decide if the app should proceed.

Background It

Background it

- Always background your network activity.
- This includes any “post” processing of the data such that the UI thread is handed the final object.
- Many solutions. Ideal will be based on apps need.

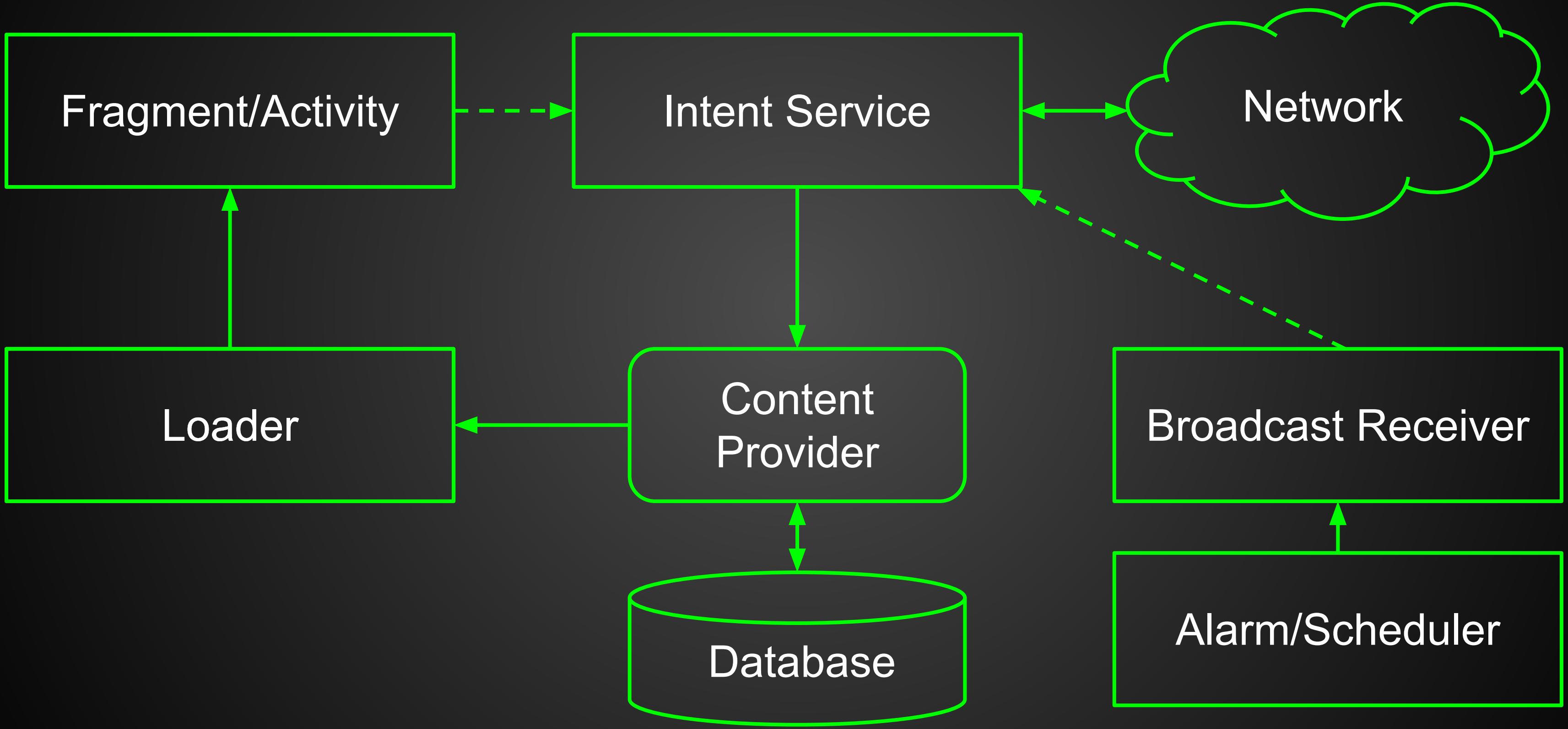
AsyncTask



Async Tasks

- Good
  - Handy if giving network data directly to the UI.
  - Useful for on demand data (bitmaps)
- Bad
  - Should be cleaned up/aborted if user quits activity
  - Must wait for network call before data is shown.
  - Special handling needed for bitmaps/lists
  - Not ideal for long running operations.

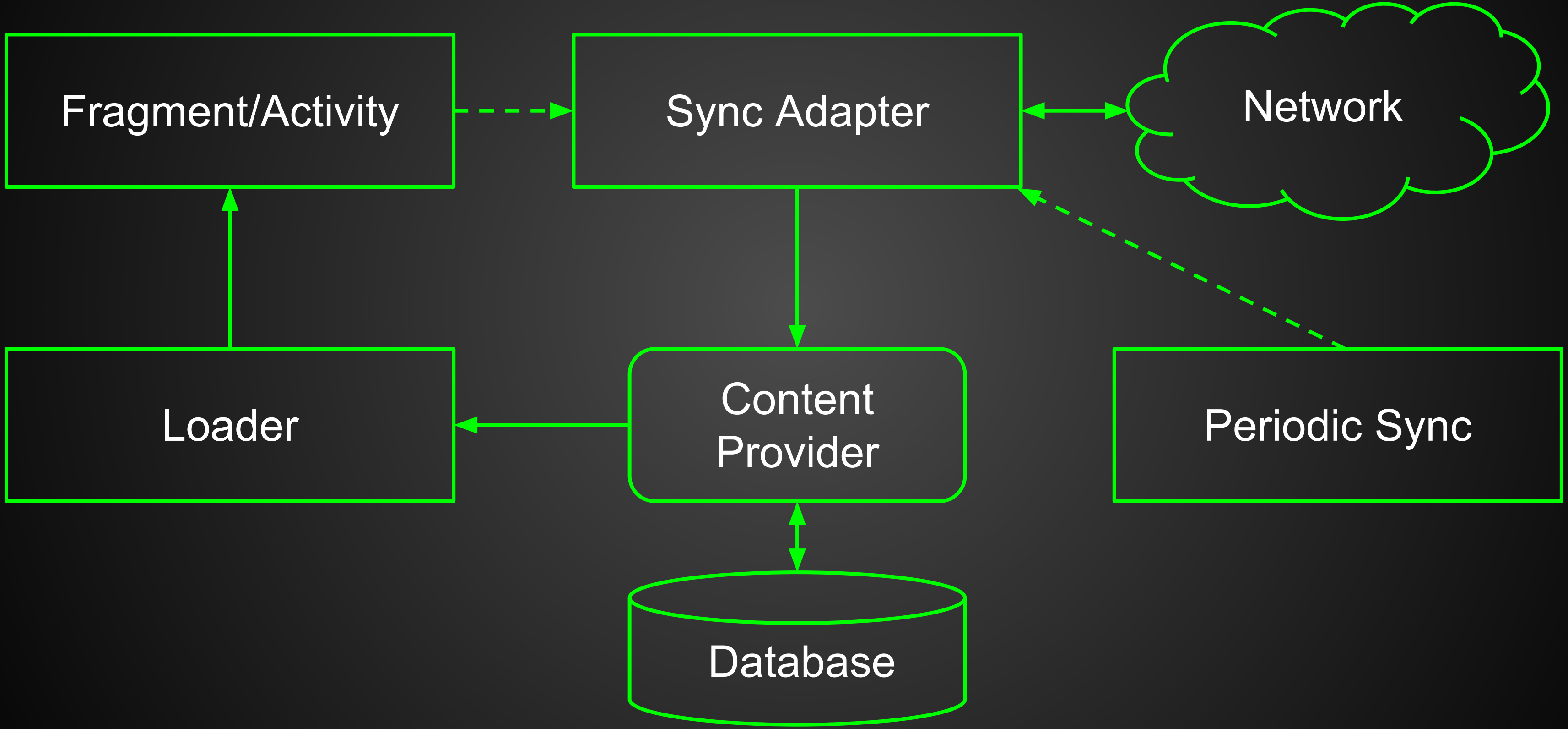
Intent Service



Intent Service

- Good
  - Handy for pre-fetching data in background.
  - Can check if data stale and load as needed.
  - Allows for batching calls into one thread.
  - UI refreshes when data load complete with loaders
- Bad
  - Will block other requests if long running.
  - Handoff needed to UI if need to present error.

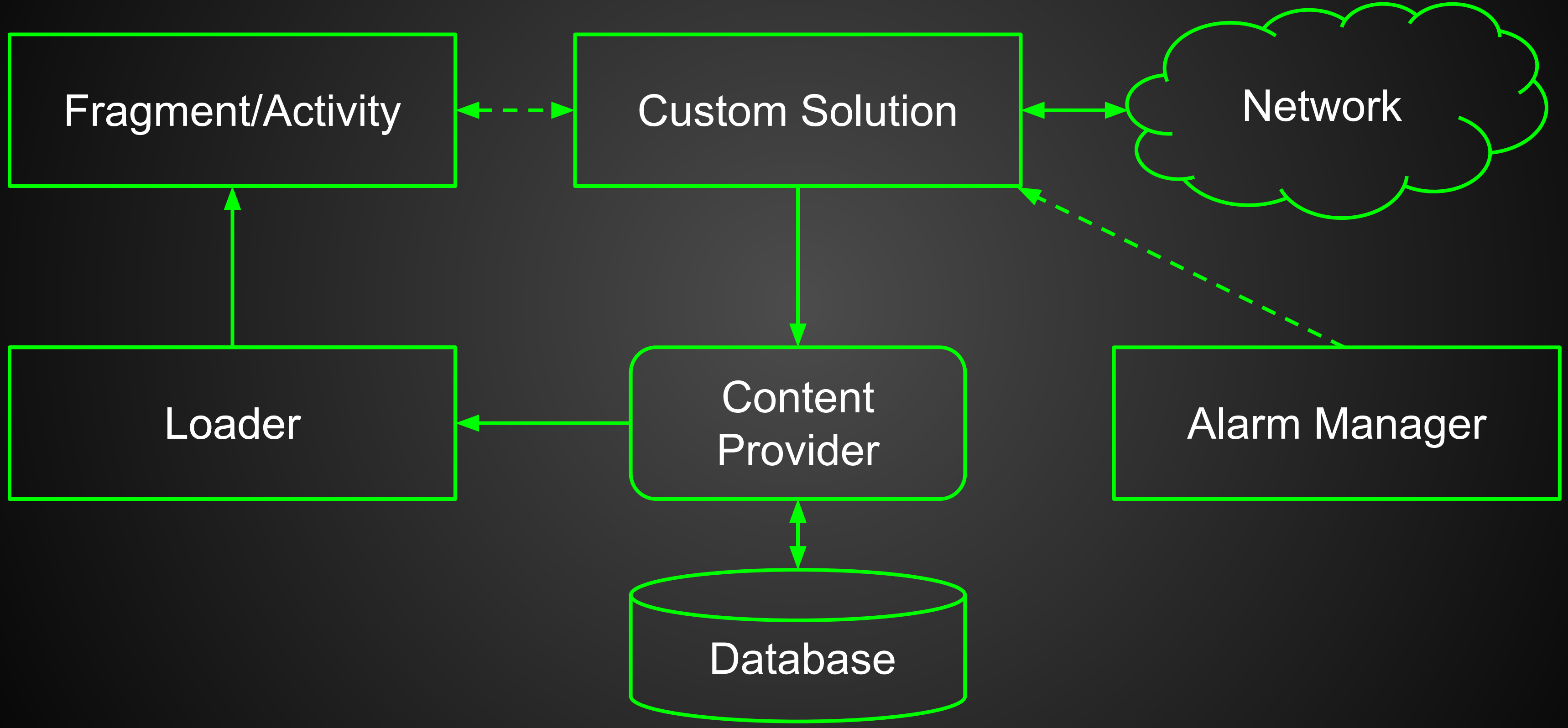
Sync Adapter



Sync Adapter

- Good
  - Similar to service, but allows the OS to plan syncs when network available and request period.
  - Can schedule to run on a periodic basis
  - Can be triggered manually
- Bad
  - More complex to set up
  - Requires an Authenticator/Account

Custom Solution



Custom Solution

- Good
  - Most customizable/flexible
  - Usually using a thread pool
  - Third party implementations
- Bad
  - Sometimes overkill
  - A lot of work to get right
  - Third party implementations

Third parties

- Image Loading
  - Picasso, Glide
- Data loading
  - Volley, Retrofit, Ion
  - Note, haven’t used any of them. Had my own variation used until now.

Connectivity Handling

Check to see if network is available

- Use the Connectivity Manager to see if network is available before issuing requests that fail.
- No connectivity may mean user is out of range of a cell tower or wifi range, or simply forgot to turn off airplane mode.
- Most cases user is unable to fix (except airplane mode)

Avoid alarming the user

- Unless the user can intervene, try not to alarm the user with network failure or connectivity issues.
- Ideally app is able to “pre-fetch” or use cached data to allow app to function when connectivity is missing.
- Detect a connectivity change and restart the load.. (more details next week).

Exponential Backoff

- Use exponential backoff to retry failures.
  - Unless there is a bug, failures are likely intermittent.
  - Have a cap on when to “quit” however if server is completely offline.
  - Understand the API responses. Some may indicate response is “in progress” and to try again.
  - Ideally add a “random” component so not every copy of app is retrying at same frequencies.

Making the connection

Form the URL

- URLs have specific escape codes especially query parameters.
  - Don’t assume you will always get it right.
- Use Uri.Builder() to form the url string which will properly encode the URL.
- Create the URL from the string emitted by the Uri.toString() method.

Creating the URL connection

- Once you have the URL, you just call openConnection() to create the connection for all other operations.
- You should cast to an HttpURLConnection to expose the http api.
- While the call implies a connection, the connection is not actually open yet...

Preparing the connection

- All steps need to happen before connecting.
  - Choose the HTTP method. GET, POST, etc.
  - Configure any connection and read timeouts.
  - Add any request headers the app needs to apply.
  - If using POST/PUT, etc and sending data, call setDoOutput(true) to request that an output stream be created.

PUT/POST performance

- To ensure optimum performance, you should call either setFixedLengthStreamingMode() or setChunkedStreamingMode() to indicate if the app knows the size of the data being sent.
- Failure to call either method results in the entire request being loaded into memory before being sent increasing memory use and/or latency.
- Use a chunk size of 0 to indicate you want the default chunk size.

Call connect/disconnect

- Call connect when ready to start the connection. At this point the connection will be opened and any streams will be created.
- Place most of the logic in a try/finally block. In the finally block call disconnect() to return the connection to the pool or close it. The OS will decide its fate.

Streams

- Use buffered input/output streams to read/write data to the connection. The raw streams are not buffered.
- Ideally always process data from directly streams into the "final" objects.
    - Bitmaps, JSONReader, XMLPullParser all support streamed data.
    - If storing into a database, read into content values.

Streams cont.

- ErrorStream
    - Errors not sent via input stream. Special stream to read errors.

Resources

Resources

- HttpURLConnection
- HttpResponseCache
- Updating Security Provider
- Android Http Clients (2011)
- HTTP RFC2616