



Hugbúnaðarverkefni 2 / Software Project 2

10. Networking and Threading in Android

HBV601G – Spring 2019

Matthias Book



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRUVÍSINDASVIÐ
IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

In-Class Quiz 8 Prep

- Please prepare a small scrap of paper with the following format:

ID: _____@hi.is Date: _____

a) _____ e) _____
b) _____ f) _____
c) _____ g) _____
d) _____ h) _____

- During class, I'll show you questions that you can answer with some letters
- Hand in your scrap at end of class
- All questions in a quiz have same weight
- All quizzes (8-10 throughout semester) have the same weight
 - Your worst 2 quizzes will be disregarded
- Overall quiz grade counts as optional question worth 7.5% on final exam



Recap: Where to Store Your App's Data?

- Static media for integration in your app's user interface (e.g. images)
 - Resource directory of your project
- Volatile short-term storage of little data while activity is inactive (ie. activity state)
 - `savedInstanceState` bundle of an activity
- Persistent storage of simple data that only your app works with (e.g. game state)
 - `SharedPreferences` file in your app's private internal storage directory
- Persistent storage of large data that only your app works with (e.g. cached data)
 - Files in your app's private internal storage directory
- Dynamic, unstructured data that you exchange with other apps (e.g. documents)
 - Files in the user's external storage directories
- Dynamic, structured data that only your app works with (e.g. a to-do list)
 - Local SQLite database in your app's internal storage directory
- Data that you want to share with other users or devices (e.g. messages)
 - Central data storage on a remote server (*today's class*)

Recap: Internal vs. External Storage

Internal Storage

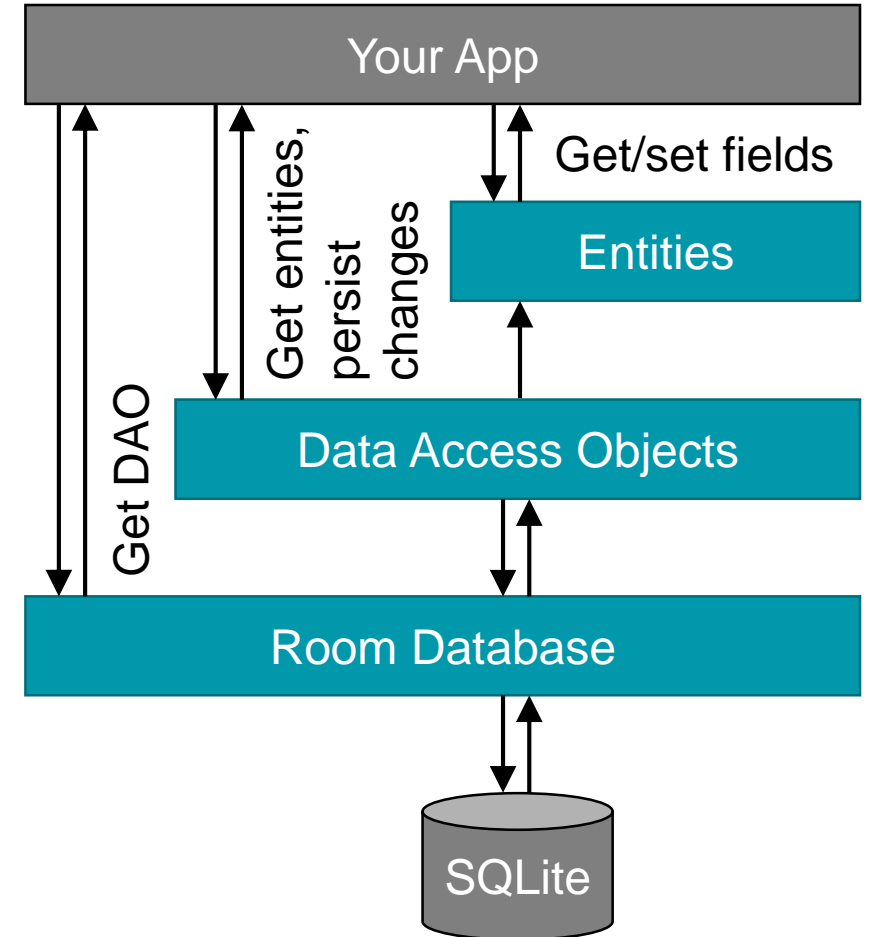
- Built-in, always available
 - Files are accessible only by your app
 - Does not require special permission
 - All files deleted upon uninstalling app
- **Suitable for files that should be access-restricted**
- e.g. app state, sensitive data

External Storage

- Might be SD card unmounted by user
 - highly unlikely in modern devices
 - Files accessible by all of user's apps
 - Requires permission upon installation
 - “Private” files deleted upon uninstalling app, “public” files remain
- **Suitable for files that are intended for sharing/outliving your app**
- e.g. created media/documents

Recap: Room Persistence Library Architecture

- **@Database** provides an abstraction of the underlying SQLite database containing the app's persisted relational data
- **@Dao** instances contain methods for getting entities from the database and saving changes back to the database
- **@Entity** instances represent tables within the database and provide access to its columns



Client-Server Communication

using HTTP and JSON

see also:

- Phillips et al.: Android Development, Ch. 23-24 (2nd ed.) / 25-26 (3rd ed.)
- <https://developer.android.com/training/basics/network-ops/connecting.html>



Overview

- The networking part of client-server communication in Android is implemented in the same way as in any other networked application.
 - First part of this class
- Namely, we will use:

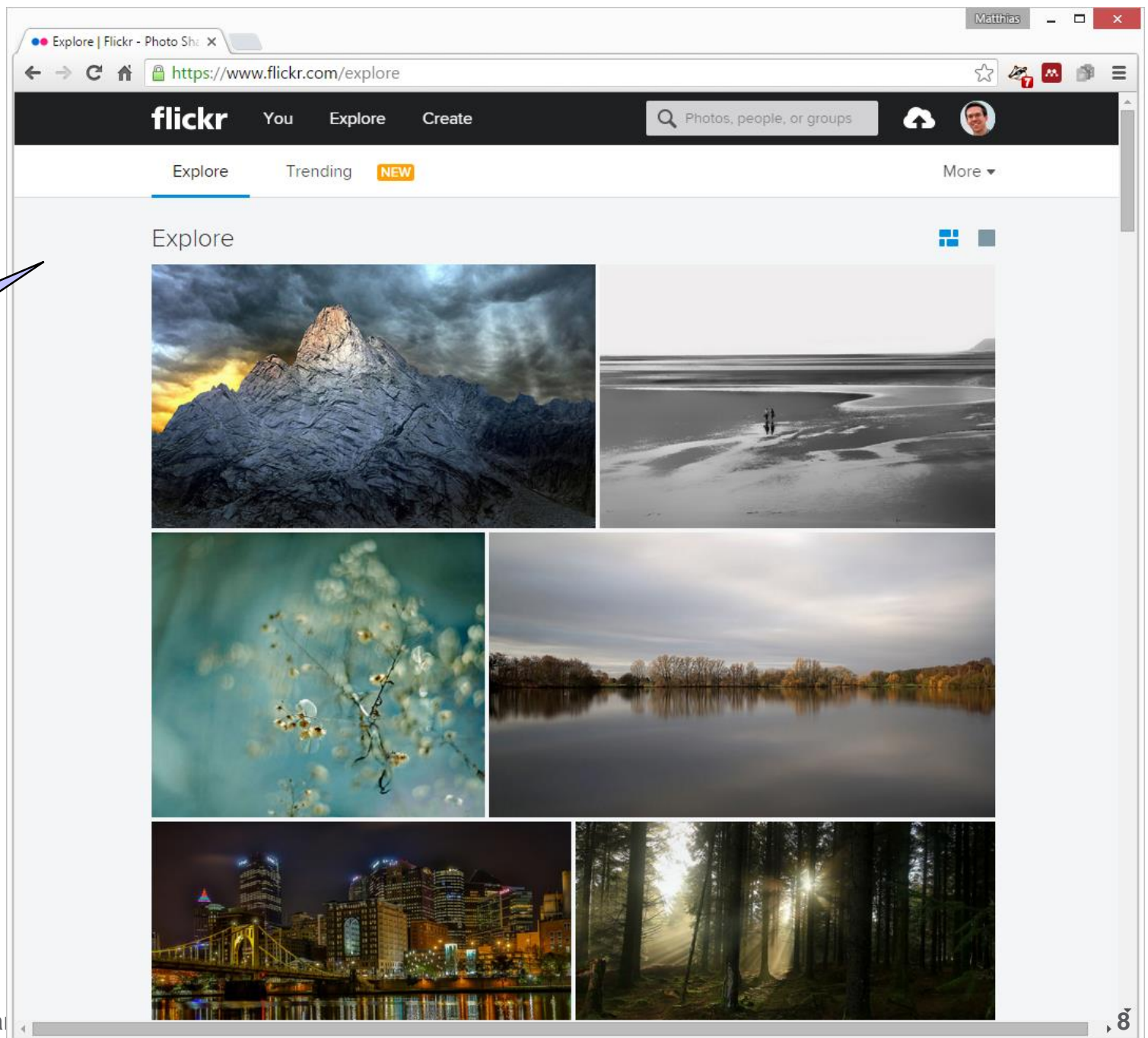
```
import java.io.ByteArrayOutputStream;  
import java.io.InputStream;  
import java.net.HttpURLConnection;  
import java.net.URL;
```
- Recommended to review the Java API documentation for these classes
- Things get more complex because of Android-specific multi-threading mechanics though.
 - Second part of this class

Example: A Flickr Viewer



Our goal

Our data
source



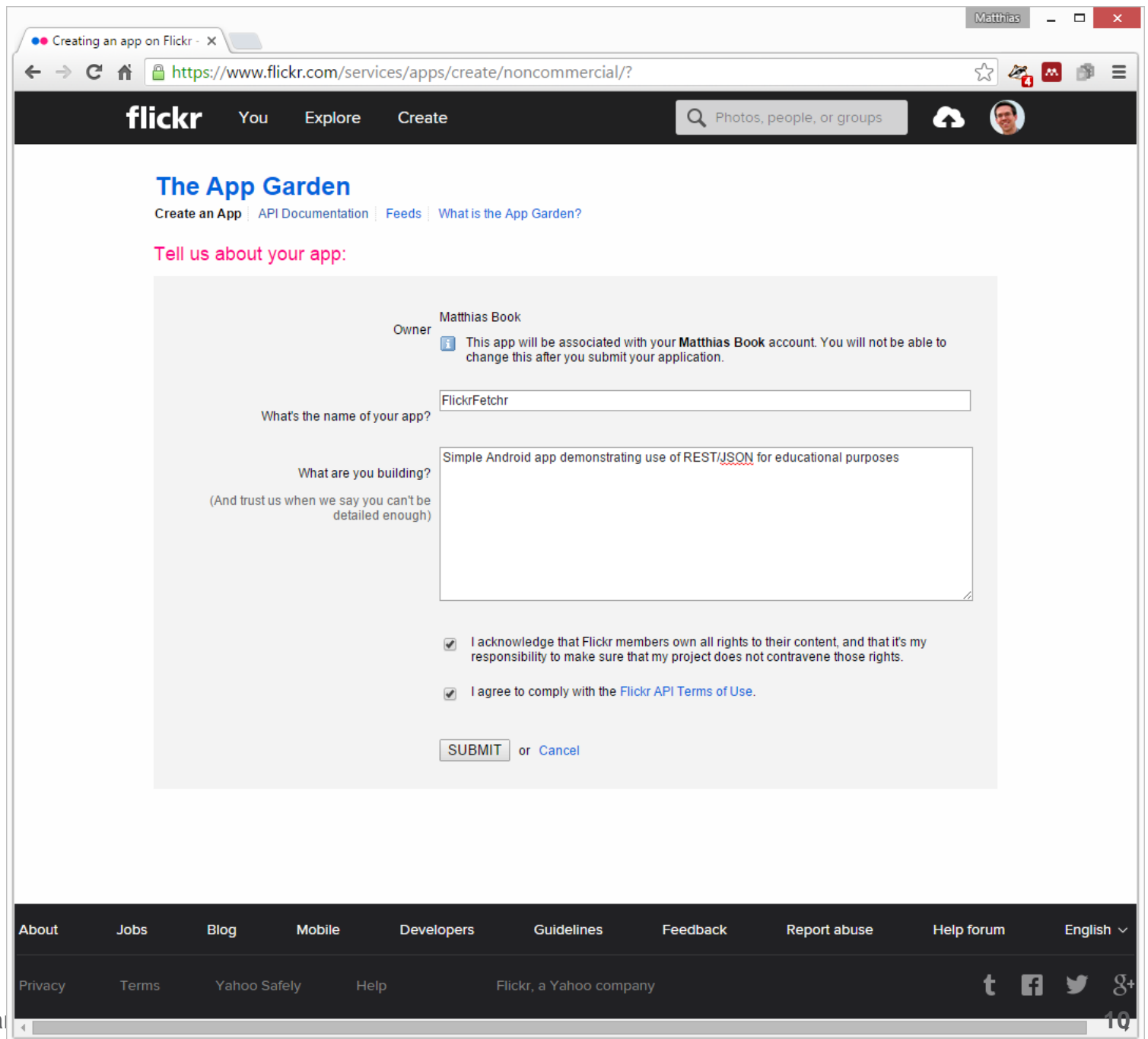
Prep: Flickr API Documentation

- Flickr provides a rich set of API methods to retrieve data from their site
 - www.flickr.com/services/api/
- You'll need an API key to access the Flickr API...
- ...and familiarize yourself with their REST interface

The screenshot shows the Flickr API documentation page. The browser tab is titled 'Flickr Services' and the address bar shows 'https://www.flickr.com/services/api/'. The page has a dark header with the Flickr logo and navigation links: 'You', 'Explore', 'Create'. A search bar contains the text 'Photos, people, or groups'. The main content area is titled 'The App Garden' and includes links for 'Create an App', 'API Documentation', 'Feeds', and 'What is the App Garden?'. A text block states: 'The Flickr API is available for non-commercial use by outside developers. Commercial use is possible by prior arrangement.' Below this is a section 'Read these first:' with a list of links: 'Developer Guide', 'Overview', 'Encoding', 'User Authentication', 'Dates', 'Tags', 'URLs', 'Buddyicons', and 'Flickr APIs Terms of Use'. The 'API Keys' link is circled in red, with a red arrow pointing from the text 'You'll need an API key to access the Flickr API...'. Below this is the 'Photo Upload API' section with links: 'Uploading Photos', 'Replacing Photos', 'Example Request', and 'Asynchronous Uploading'. The 'Request Formats' section has a list: 'REST', 'XML-RPC', and 'SOAP'. The 'REST' link is circled in red, with a red arrow pointing from the text '...and familiarize yourself with their REST interface'. The 'Response Formats' section has a list: 'REST', 'XML-RPC', 'SOAP', 'JSON', and 'PHP'. On the right side, there is a 'API Methods' section with various categories and their corresponding API methods: 'activity' (flickr.activity.userComments, flickr.activity.userPhotos), 'auth' (flickr.auth.checkToken, flickr.auth.getFrob, flickr.auth.getFullToken, flickr.auth.getToken), 'auth.oauth' (flickr.auth.oauth.checkToken, flickr.auth.oauth.getAccessToken), 'blogs' (flickr.blogs.getList, flickr.blogs.getServices, flickr.blogs.postPhoto), 'cameras' (flickr.cameras.getBrandModels, flickr.cameras.getBrands), 'collections' (flickr.collections.getInfo, flickr.collections.getTree), 'commons' (flickr.common.getInstitutions), and 'contacts'.

Prep: Obtaining a Flickr API Key

- Request a non-commercial API key



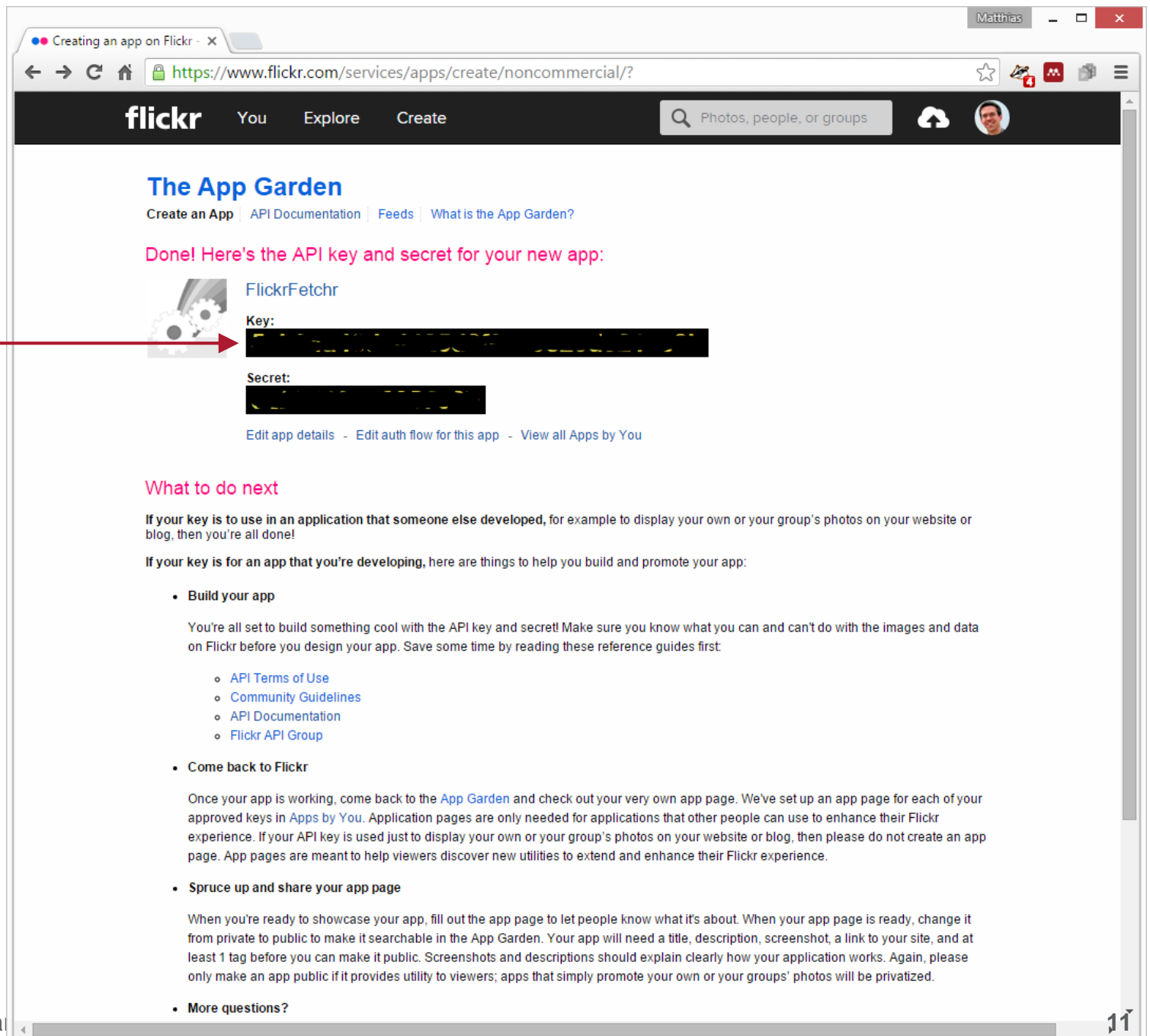
The screenshot shows the Flickr 'The App Garden' page for creating a non-commercial app. The browser address bar shows the URL: <https://www.flickr.com/services/apps/create/noncommercial/>. The page title is 'The App Garden' with links for 'Create an App', 'API Documentation', 'Feeds', and 'What is the App Garden?'. The main heading is 'Tell us about your app:'. The form fields are as follows:

- Owner:** Matthias Book. A note states: 'This app will be associated with your **Matthias Book** account. You will not be able to change this after you submit your application.'
- What's the name of your app?:** FlickrFetchr
- What are you building?:** Simple Android app demonstrating use of REST/JSON for educational purposes. (And trust us when we say you can't be detailed enough)
- Agreements:** Two checked checkboxes: 'I acknowledge that Flickr members own all rights to their content, and that it's my responsibility to make sure that my project does not contravene those rights.' and 'I agree to comply with the Flickr API Terms of Use.'
- Buttons:** SUBMIT or Cancel

The footer contains links for About, Jobs, Blog, Mobile, Developers, Guidelines, Feedback, Report abuse, Help forum, English, Privacy, Terms, Yahoo Safety, Help, and Flickr, a Yahoo company.

Prep: Obtaining a Flickr API Key

- You'll just need the key for this project, not the secret



Creating an app on Flickr - X

https://www.flickr.com/services/apps/create/noncommercial/?

flickr You Explore Create

Photos, people, or groups

The App Garden

Create an App | API Documentation | Feeds | What is the App Garden?

Done! Here's the API key and secret for your new app:

FlickrFetchr

Key: [REDACTED]

Secret: [REDACTED]

Edit app details - Edit auth flow for this app - View all Apps by You

What to do next

If your key is to use in an application that someone else developed, for example to display your own or your group's photos on your website or blog, then you're all done!

If your key is for an app that you're developing, here are things to help you build and promote your app:

- Build your app

You're all set to build something cool with the API key and secret! Make sure you know what you can and can't do with the images and data on Flickr before you design your app. Save some time by reading these reference guides first:

- [API Terms of Use](#)
- [Community Guidelines](#)
- [API Documentation](#)
- [Flickr API Group](#)

- Come back to Flickr

Once your app is working, come back to the [App Garden](#) and check out your very own app page. We've set up an app page for each of your approved keys in [Apps by You](#). Application pages are only needed for applications that other people can use to enhance their Flickr experience. If your API key is used just to display your own or your group's photos on your website or blog, then please do not create an app page. App pages are meant to help viewers discover new utilities to extend and enhance their Flickr experience.

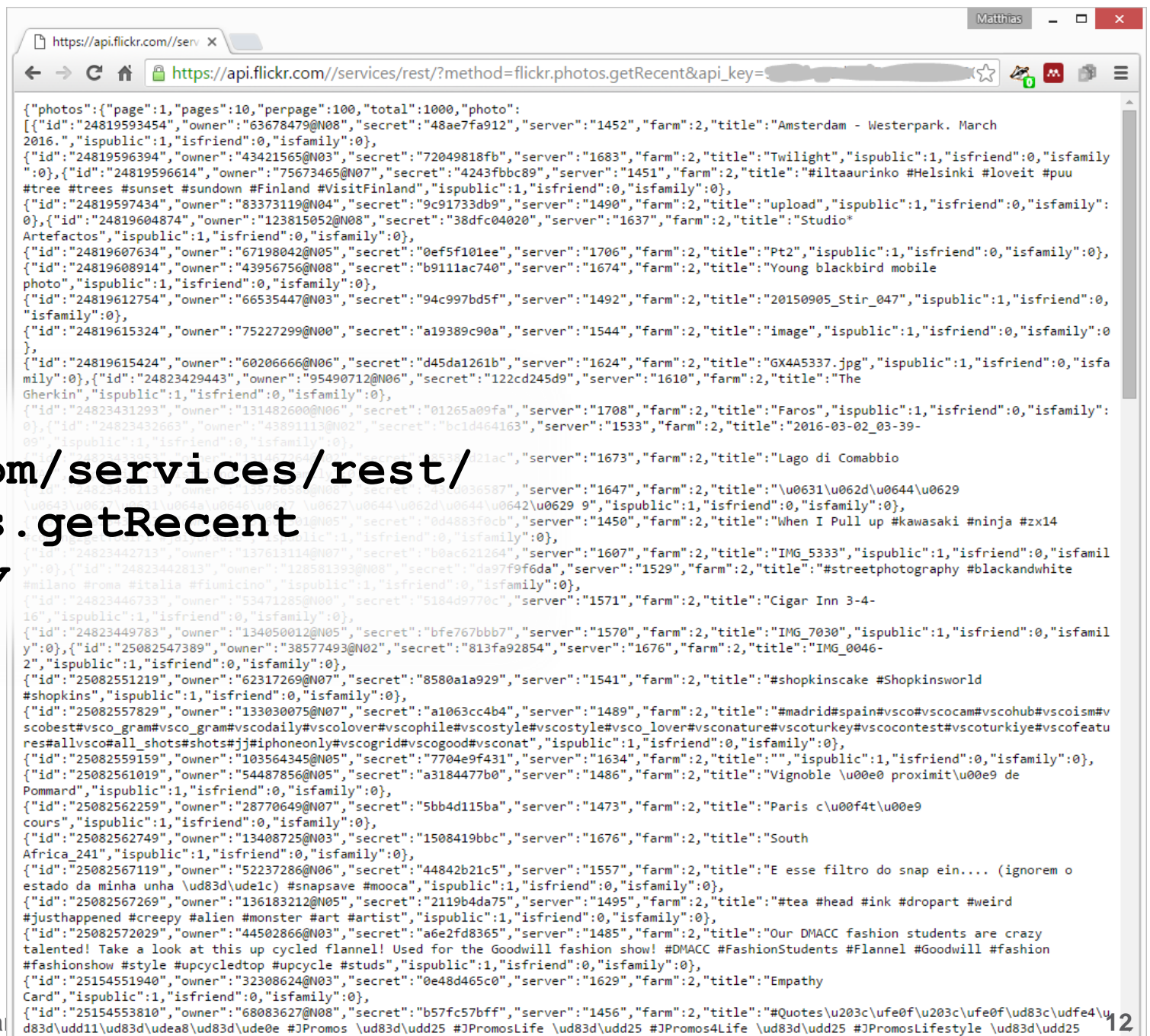
- Spruce up and share your app page

When you're ready to showcase your app, fill out the app page to let people know what it's about. When your app page is ready, change it from private to public to make it searchable in the App Garden. Your app will need a title, description, screenshot, a link to your site, and at least 1 tag before you can make it public. Screenshots and descriptions should explain clearly how your application works. Again, please only make an app public if it provides utility to viewers; apps that simply promote your own or your groups' photos will be privatized.

- More questions?

Raw JSON Output of the Flickr API

- Example: Retrieve list of recently uploaded pictures in JSON format
- `https://api.flickr.com/photos/your_username/?method=flickr.photos.getRecent&api_key=your_API_key&format=json&nojsoncallback=1`
- This is the source format our app will work with.



Getting Permission to Network

- Network access from an Android app requires user permission
- This can be requested at the time of installation by including the following lines in `/app/src/main/AndroidManifest.xml`:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.bignerdranch.android.photogallery" >

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

    <application
        ...
    </application />
</manifest>
```


Reading Any Data via HTTP (FlickrFetchr.java)

```
public class FlickrFetchr {
    private static final String TAG = "FlickrFetchr";
    private static final String API_KEY = "your_API_key";
    public byte[] getUrlBytes(String urlSpec) throws IOException {
        URL url = new URL(urlSpec);
        HttpURLConnection conn = (HttpURLConnection)url.openConnection();
        try {
            ByteArrayOutputStream out = new ByteArrayOutputStream();
            InputStream in = conn.getInputStream();
            if (conn.getResponseCode() != HttpURLConnection.HTTP_OK) {
                throw new IOException(conn.getResponseMessage() + ": with " + urlSpec);
            }
            int bytesRead = 0;
            byte[] buffer = new byte[1024];
            while ((bytesRead = in.read(buffer)) > 0) {
                out.write(buffer, 0, bytesRead);
            }
            out.close();
            return out.toByteArray();
        } finally { conn.disconnect(); }
    }
    public String getUrlString(String urlSpec) throws IOException {
        return new String(getUrlBytes(urlSpec));
    }
    // ...
}
```

Open HTTP connection to service at given URL

Prepare to store data

Request data from URL

Check request success

Read 1024-byte chunks from HTTP response and write them into output stream

Create byte array from output stream contents

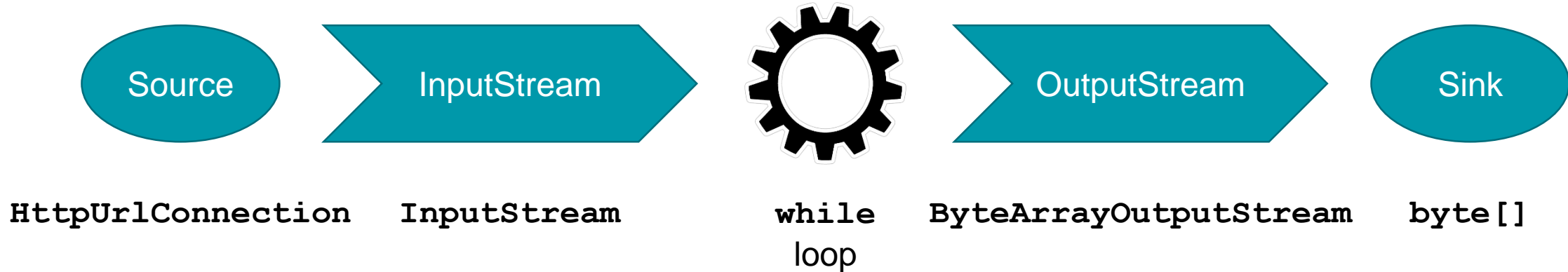
Close output stream

Close network connection, whatever happens

Interpret contents of the byte array as a String

Nothing specific here – can use this in any app

Excursion: Working with Streams



- Java class library provides many more stream types for various operations...
 - Reading from / writing to files, network connections, threads...
 - Reading / writing bytes, strings, objects...
 - Buffering, formatting, encrypting / decrypting, compressing / decompressing...
- ...which can be combined arbitrarily!
- See <https://docs.oracle.com/javase/tutorial/essential/io/streams.html>
- and the Java API documentation for the various subclasses of `java.io.InputStream` and `java.io.OutputStream`

Fetching the List of Recent Pictures from Flickr (FlickrFetchr.java)

```
public List<GalleryItem> fetchItems() {  
    List<GalleryItem> items = new ArrayList<>();  
    try {  
        String url = Uri.parse("https://api.flickr.com/services/rest/")  
            .buildUpon()  
            .appendQueryParameter("method", "flickr.photos.getRecent")  
            .appendQueryParameter("api_key", API_KEY)  
            .appendQueryParameter("format", "json")  
            .appendQueryParameter("nojsoncallback", "1")  
            .appendQueryParameter("extras", "url_s")  
            .build().toString();  
  
        String jsonString = getUrlString(url);  
        Log.i(TAG, "Received JSON: " + jsonString);  
        JSONObject jsonBody = new JSONObject(jsonString);  
        parseItems(items, jsonBody);  
    } catch (IOException ioe) {  
        Log.e(TAG, "Failed to fetch items", ioe);  
    } catch (JSONException je) {  
        Log.e(TAG, "Failed to parse JSON", je);  
    }  
    return items;  
}
```

Data structure in which we will collect the fetched data

Construct a URL with query parameters

Call our previously shown method to retrieve a response string from a URL

Pass the string to an object that provides methods for handling JSON data

A method we'll implement to parse the JSON array and create a list of `GalleryItem` objects from it

The Photo Gallery's Data Model (GalleryItem.java)

- This is the destination format for the data our app receives from the server:

```
public class GalleryItem {
```

```
    private String mCaption;  
    private String mId;  
    private String mUrl;
```

Of the data provided in the JSON response, we'll store just the caption, URL and Flickr ID of each image

```
    public String getCaption() { return mCaption; }  
    public void setCaption(String caption) { mCaption = caption; }
```

```
    public String getId() { return mId; }  
    public void setId(String id) { mId = id; }
```

```
    public String getUrl() { return mUrl; }  
    public void setUrl(String url) { mUrl = url; }
```

@Override

```
    public String toString() { return mCaption; }
```

```
}
```



Parsing JSON Data into Model Objects (FlickrFetchr.java)

```
private void parseItems(List<GalleryItem> items, JSONObject jsonBody)
    throws IOException, JSONException {

    JSONObject photosJsonObject = jsonBody.getJSONObject("photos");
    JSONArray photoJsonArray = photosJsonObject.getJSONArray("photo");

    for (int i = 0; i < photoJsonArray.length(); i++) {
        JSONObject photoJsonObject = photoJsonArray.getJSONObject(i);

        GalleryItem item = new GalleryItem();

        item.setId(photoJsonObject.getString("id"));
        item.setCaption(photoJsonObject.getString("title"));
        if (!photoJsonObject.has("url_s")) {
            continue;
        }
        item.setUrl(photoJsonObject.getString("url_s"));

        items.add(item);
    }
}
```

Obtain the JSON array
holding photo references

For each JSON array
element...

...construct a new
GalleryItem,
populate its attributes...

...and add it to
the list of **items**

Retrieving Bitmaps via HTTP

(ThumbnailDownloader.java)

- Retrieving the image files that the `GalleryItems` refer to is quite easy:
 1. We use the method `byte[] getUrlBytes(String url)` that we implemented in `FlickrFetchr` to retrieve the binary data for the file (e.g. a JPG image)
 2. We use the static method `Bitmap decodeByteArray(byte[] data, int offset, int length)` of `android.graphics.BitmapFactory` to convert the image data into a `Bitmap` object that can be incorporated into our user interface.
- Example:

```
byte[] bitmapBytes = new FlickrFetchr().getUrlBytes(url);  
final Bitmap bitmap = BitmapFactory.decodeByteArray(bitmapBytes, 0, bitmapBytes.length);
```
- Transfer-wise, this is all we need to retrieve and parse JSON and binary data...
- ...but when using these methods in a mobile app, we still need to consider that network communication is *slow*.

Background Tasks

using `AsyncTask`

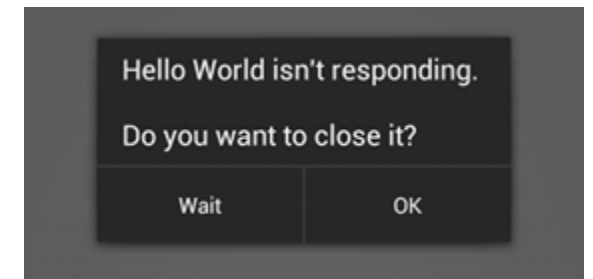
see also:

- Phillips et al.: Android Development, Ch. 23 (2nd ed.) / 25 (3rd ed.)
- <https://developer.android.com/training/articles/perf-anr.html>
- <https://developer.android.com/reference/android/os/AsyncTask.html>



The Main/UI Thread

- The Activities of each Android app are executed in the app's **main thread**
 - The main thread (also called **UI thread**) is an infinite loop that
 - waits for events from the user or system (e.g. Activity startups, button taps etc.)
 - and calls the respective event handlers
 - If execution of any event handler takes too long, the UI becomes unresponsive
 - If an app doesn't respond to a UI event within 5 seconds, the Android OS will offer to close it
- Anything that has the potential to take a long time, e.g...
- any network operations
 - file operations that may be blocked due to concurrent access
 - high-complexity or high-volume algorithms
- ...should be executed in a different thread than the main thread
 - Android actually prevents network operations on main thread (**NetworkOnMainThreadException**)



"App not responding"
(ANR) dialog

Background Threads

- There are several mechanisms for executing work in background threads (i.e. other threads than the app's UI thread):
 - Android provides one background thread for “light”, short-lived, infrequent tasks tied to a particular activity (so called **AsyncTasks**) (*→ following slides; Android textbook, Ch. 23*)
 - All **AsyncTasks** are executed sequentially in the app's same background thread
 - [Dis]advantage: You cannot/need not manage the tasks' threading mechanics
 - You can create your own background threads for “heavy”, long-running tasks that are tied to a particular activity (so called **HandlerThreads**) (*→ Android textbook, Ch. 24*)
 - Each **HandlerThread** is executed in a background thread of its own, controlled by an activity
 - [Dis]advantage: You can/need to manage the threading mechanics yourself
 - You can create your own background threads for any tasks that are executed independently from activities (so called **IntentServices**) (*→ end of this class; Android textbook, Ch. 26*)
 - Each **IntentService** is executed in a thread of its own, started via an intent
 - Advantages: Completely independent from an app's UI; can be scheduled

Invocation of an AsyncTask (PhotoGalleryFragment.java)

```
public class PhotoGalleryFragment extends Fragment {  
  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setRetainInstance(true);  
        AsyncTask task = new FetchItemsTask();  
        task.execute();  
        // ...  
    }  
  
    // ...  
}
```

Keep this fragment object intact even if its hosting activity is re-created (e.g. due to device rotation)

So we don't need to re-create the **AsyncTask**

Create the new task and tell the background thread to execute it

Note: **AsyncTasks** in background thread are queued, so execution may not begin immediately!

Implementation of an AsyncTask (PhotoGalleryFragment.java)

```
public class PhotoGalleryFragment extends Fragment {
```

```
    @Override
```

```
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setRetainInstance(true);  
        AsyncTask task = new FetchItemsTask();  
        task.execute();  
        // ...  
    }
```

```
    // ...
```

Make this an
AsyncTask

```
    private class FetchItemsTask extends AsyncTask<Void, Void, List<GalleryItem>> {  
        @Override  
        protected List<GalleryItem> doInBackground(Void... params) {  
            return new FlickrFetchr().fetchItems();  
        }  
    }
```

Type of
input data

Type of
output data

Q: Where does doInBackground's
return value end up?

The task.execute call is in the
wrong place and in a different thread ☹️

In this case, calling our method
that retrieves the JSON string
from Flickr's REST API

doInBackground contains
the code to be executed in
the background thread

Returning Data from an AsyncTask to the Main Thread (PhotoGalleryFragment.java)

```
public class PhotoGalleryFragment extends Fragment {  
  
    private List<GalleryItem> mItems = new ArrayList<>();  
  
    // ...  
  
    private class FetchItemsTask extends AsyncTask<Void, Void, List<GalleryItem>> {  
  
        @Override  
        protected List<GalleryItem> doInBackground(Void... params) {  
            return new FlickrFetchr().fetchItems();  
        }  
  
        @Override  
        protected void onPostExecute(List<GalleryItem> items) {  
            mItems = items;  
            setupAdapter();  
        }  
    }  
}
```

A: `doInBackground`'s return value is received by `AsyncTask`'s private implementation and passed to the task's `onPostExecute` method...

...which is executed in the *main* instead of the *background* thread!

Copy the return value to the fragment's member variable

Takes care of displaying the `GalleryItems`

For clarity, we skip all aspects of the View layer in this example – see Ch. 9, 23 & 24 of the Android textbook and corresponding source code for details.

Optional: Progress Updates from Within an AsyncTask

```
ProgressBar mProgressBar = (ProgressBar) findViewById(R.id.progress_bar);  
// ...
```

Type of
progress data

```
AsyncTask<Void,Integer,Void> task = new AsyncTask<Void,Integer,Void>() {
```

```
    public Void doInBackground(Void... params) {  
        Integer percentComplete;  
        while (...) { // some loop condition  
            // ...background operations...  
            publishProgress(percentComplete);  
        }  
    }
```

Keep track of progress (here: in **percentComplete**) in the **doInBackground** method (executed in background thread)

```
    public void onProgressUpdate(Integer... params) {  
        int progress = params[0];  
        mProgressBar.setProgress(progress);  
    }
```

Call **publishProgress** whenever the progress indicator in the View shall be updated

```
};
```

```
// ...
```

```
task.execute();
```

...where a suitable widget can be updated

A call to **publishProgress** will trigger execution of **onProgressUpdate** in the main thread...

Cancelling an AsyncTask

- If you need to cancel an **AsyncTask** before it has begun or completed, you can call its **cancel(boolean mayInterrupt)** method. Then:
 - If the cancelled task's **doInBackground** method hasn't been called yet, it won't be called
 - If the **doInBackground** method is already active: If **mayInterrupt** is
 - **false**: the method will keep running, but the task's "cancelled" flag will be set
 - You can check this flag occasionally in your **doInBackground** implementation by calling the **boolean isCancelled()** method, and elect to finish work prematurely if you find the flag set
 - **true**: the method will be interrupted immediately – try to avoid this
 - After cancelling an **AsyncTask**, its **onCancelled** method will be called instead of its **onPostExecute** method
- When and where cancellation of an **AsyncTask** makes sense may vary:
 - Possibly cancel it in the activity's/fragment's **onStop** or **onDestroy** method
 - Possibly simply let it run to completion
 - But ensure it doesn't invoke methods on an activity that may already be in an invalid state

Background Services

using `IntentService`

see also:

- Phillips et al.: Android Development, Ch. 26 (2nd ed.) / 28 (3rd ed.)
- <https://developer.android.com/guide/components/services.html>



Activities vs. Services

- So far, we know only activities as the prime components of our Android apps
 - An **IntentService** is a component treated in some ways like an activity, i.e...
 - it can be started and destroyed
 - it is invoked via an intent
 - ...but differing in some important ways from activities, namely:
 - it has no associated user interface (view)
 - it runs in its own (background) thread
- This means that unlike **AsyncTasks** and **HandlerThreads**, **IntentServices** can perform background work that is completely independent from activities:
- A service can be active even when none of the app's activities are alive
 - A service can be automatically (e.g. periodically) invoked by the Android OS

Implementing an IntentService's Business Logic (PollService.java)

Make this an
IntentService

Factory method for an intent
("command") that other components
can use to invoke this service

```
public class PollService extends IntentService {
```

```
    public static Intent newIntent(Context context) {  
        return new Intent(context, PollService.class);  
    }
```

Core method implementing the
business logic that shall be
executed in the background

@Override

```
    protected void onHandleIntent(Intent intent) {  
        if (!isNetworkAvailableAndConnected()) { return; }  
        // ...perform the business logic...  
    }
```

If the business logic requires
network access, check if we
actually *can* network first

```
    private boolean isNetworkAvailableAndConnected() {  
        ConnectivityManager cm = (ConnectivityManager) getSystemService(CONNECTIVITY_SERVICE);  
        boolean isNetworkAvailable = cm.getActiveNetworkInfo() != null;  
        boolean isNetworkConnected = isNetworkAvailable &&  
            cm.getActiveNetworkInfo().isConnected();  
        return isNetworkConnected;  
    }
```

Recommended to use this
code in your own apps too

Network needs to be available (not the
case if user disabled network access
for background tasks) and connected

An `IntentService`'s Life: Reacting to Commands

- An intent addressed at an `IntentService` is called a **command**.
- Upon the first command, the `IntentService` starts up and puts the command in the queue.
- Further commands are added to the queue in the order they arrive.
- Meanwhile, the `IntentService` takes one command after another from the queue and calls `onHandleIntent` for each of it.
- When the queue is empty, the `IntentService` is destroyed.
- When new commands arrive, a new instance of the `IntentService` is created, and its lifecycle starts from the beginning.

Necessary Declarations in the Manifest (AndroidManifest.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.bignerdranch.android.photogallery" >

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <application
        android:allowBackup="true" android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name" android:theme="@style/AppTheme" >
        <activity
            android:name=".PhotoGalleryActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".PollService" />
    </application>
</manifest>
```

Services need to be
declared just like activities

Immediate Invocation of an IntentService

- An **IntentService** can be invoked immediately by sending it an intent (i.e. a command):

```
Intent i = PollService.newIntent(getActivity());  
getActivity().startService(i);
```

Reminder: `getActivity` is required to obtain a `Context` when calling from within a fragment. From within an activity, you could just use `this`.

- This code could be used by some method of an activity or fragment anytime they need the **IntentService** to do something.
 - Note the syntax is quite similar to how you would send an intent to an activity.
 - The behavior is quite different though: Instead of transitioning to a new activity...
- If the **IntentService** doesn't exist yet, it will be started in the background.
- If the **IntentService** is already running, the command will be added to its queue and executed when all previous commands have been handled

Delayed Invocation of an `IntentService`

- An `IntentService` can be invoked automatically by the Android OS even when none of your activities are alive, i.e. your app's process is shut down
- Using an `AlarmManager`, you can let this happen periodically or at a specific point in time.
 - Upon each scheduled **alarm**, a predefined intent will be fired to invoke your service
- **Power consumption considerations**
 - Caution: When creating periodic alarms, consider that executing an `IntentService` more frequently will drain the battery faster
 - Android implements several strategies to conserve power:
 - The interval between alarms must be at least 60 seconds
 - Alarms will be “bundled” so several services can be invoked at the same time, even if that means that the intervals between alarms won't have exactly the specified length
 - When the device is sleeping (screen off), due alarms will not be created (so as not to wake the device up for a service invocation), unless you explicitly specify otherwise

Scheduling Alarms with an AlarmManager (PollService.java)

```
public class PollService extends IntentService {  
  
    // ...  
  
    public static void setServiceAlarm(Context context, boolean isOn) {  
        Intent i = PollService.newIntent(context);  
        PendingIntent pi = PendingIntent.getService(context, 0, i, 0);  
  
        AlarmManager alarmManager = (AlarmManager)  
            context.getSystemService(Context.ALARM_SERVICE);  
  
        if (isOn) {  
            alarmManager.setInexactRepeating(AlarmManager.ELAPSED_REALTIME,  
                SystemClock.elapsedRealtime(), AlarmManager.INTERVAL_FIFTEEN_MINUTES, pi);  
        } else {  
            alarmManager.cancel(pi);  
            pi.cancel();  
        }  
    }  
}
```

Typically invoked by
some activity or fragment

Create an intent that should later
be used to invoke our service

Only one
PendingIntent
per Intent!

Wrap intent i into a
PendingIntent

Obtain the system's
AlarmManager

Create an alarm (i.e. fire
the PendingIntent)
approx. every 15 min.

Cancel the alarm and
intent if the user wants to
stop invoking the service

Only one alarm
schedule per
PendingIntent!

Inexact Scheduling of Alarms with an AlarmManager

- One-time and repeating alarms are scheduled with **AlarmManager**'s methods

```
void set                                (int type, long trigger,                                PendingIntent operation)
void setInexactRepeating(int type, long trigger, long interval, PendingIntent operation)
```
- **trigger** indicates the time of the first alarm
 - For **type==ELAPSED_REALTIME**, milliseconds (ms) since booting the device (incl. sleep)
 - For **type==RTC**, a particular time (in ms) – Caution: UTC timezone, not adjusted for locale!
- **interval** indicates the interval between subsequent alarms
 - Use one of provided **INTERVAL_** constants (e.g. **_HOUR**) or a custom value (in ms)
- Note: Scheduling is inexact in order to conserve battery!
 - First alarm will not go off before scheduled trigger, but may go off sometime later
 - Next alarms will not go off before expiry of interval, but may go off almost a full interval later
 - Alarms will not go off if device is asleep, but be postponed until other reason for wake-up, unless **_WAKEUP** is appended to **type** constant (e.g. **RTC_WAKEUP**)

More Exact Scheduling Mechanisms

- **AlarmManager** also provides mechanisms that suggest exact scheduling (e.g. `setRepeating`), but since API level 19, these also use inexact scheduling.
- For more precise control over when your alarms occur, you can use `setWindow` (`int type`, `long windowStartMillis`, `long windowLengthMillis`, `PendingIntent operation`) to narrow the time window in which the alarm must go off.
- **AlarmManager** is just intended to fire intents. For more lightweight and precise timing operations (ticks, timeouts etc.), **Handler** is easier and more efficient.

Checking if an Alarm is Scheduled for a PendingIntent (PollService.java)

```
public static boolean isServiceAlarmOn(Context context) {  
    Intent i = PollService.newIntent(context);  
    PendingIntent pi = PendingIntent  
        .getService(context, 0, i, PendingIntent.FLAG_NO_CREATE);  
    return pi != null;  
}
```

Indicates if we have done this before, i.e. if we have scheduled an alarm

Try to wrap the intent in a **PendingIntent** again (remember: only possible once)

Return the **PendingIntent** if it already exists, or **null** if it doesn't exist

Default behavior (parameter 0) would be: Return the **PendingIntent** if it already exists, or create it if it doesn't

Interacting with the User from Within a Service

- Since services do not have associated View components (i.e. no UI), the user will (and should) normally not notice that they are active.
- When a service wants to make itself aware to the user, it has two options:
 - Send an intent that will invoke an activity, which will come to the foreground
 - Create a notification that will appear in the notifications drawer (accessible by swiping down from top of screen)
 - Often, notifications are wired up such that tapping on the notification will fire an intent that invokes a corresponding activity

Sending a Notification from a Service

(PollService.java)

@Override

```
protected void onHandleIntent(Intent intent) {  
    // ...business logic...
```

```
    Resources resources = getResources();  
    Intent i = PhotoGalleryActivity.newIntent(this);  
    PendingIntent pi = PendingIntent.getActivity(this, 0, i, 0);
```

```
    Notification notification = new NotificationCompat.Builder(this)  
        .setTicker(resources.getString(R.string.new_pictures_title))  
        .setSmallIcon(android.R.drawable.ic_menu_report_image)  
        .setContentTitle(resources.getString(R.string.new_pictures_title))  
        .setContentText(resources.getString(R.string.new_pictures_text))  
        .setContentIntent(pi)  
        .setAutoCancel(true)  
        .build();
```

```
    NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);  
    notificationManager.notify(0, notification);
```

```
    // ...business logic...
```

HÁSKÓLI ÍSLANDS

Matthias Book: Software Project 2

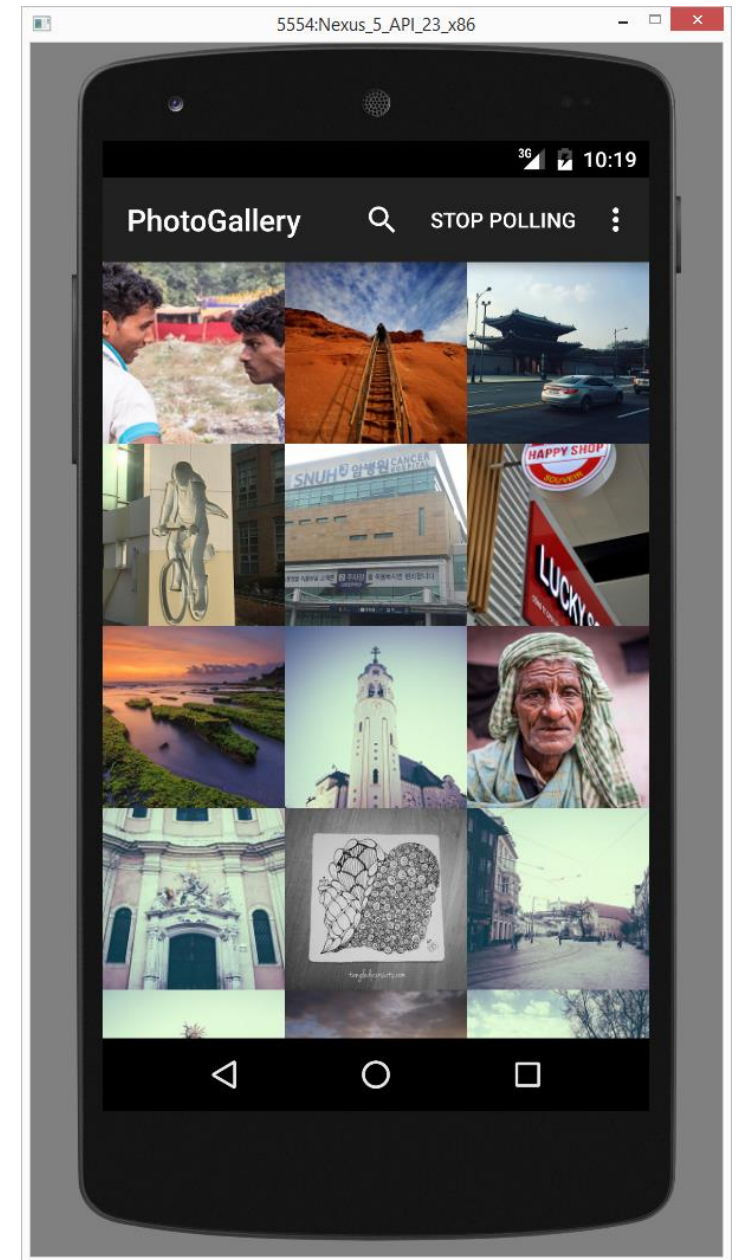
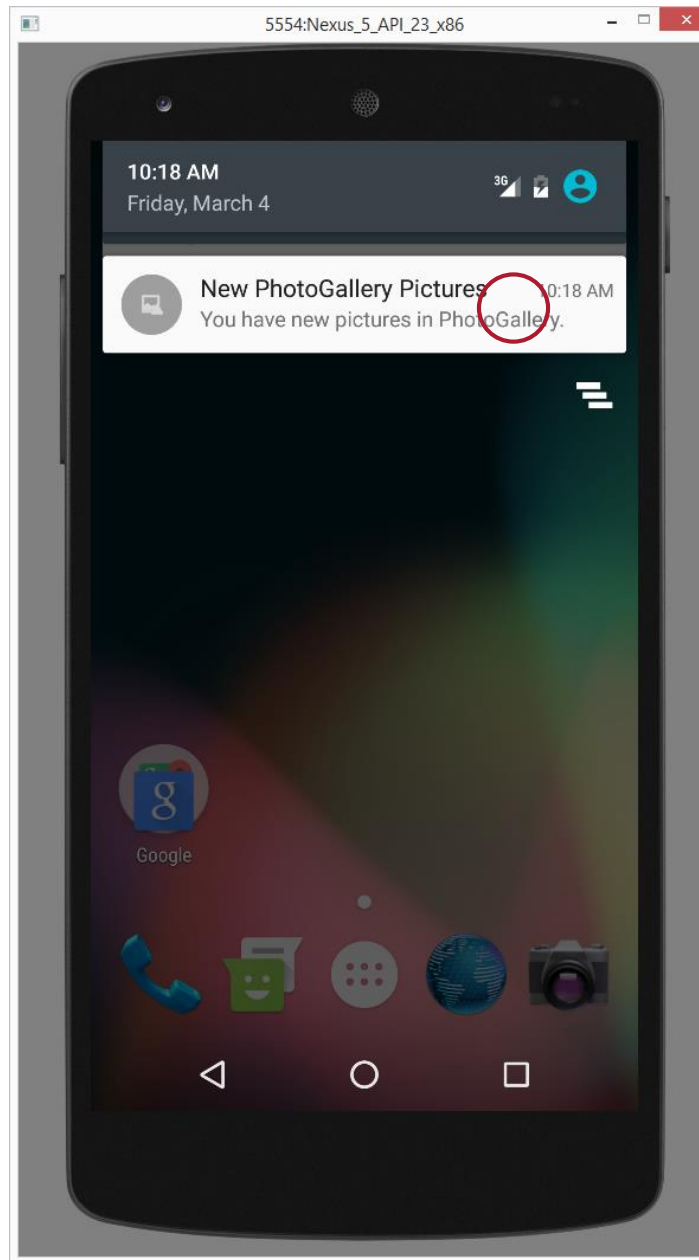
Create and wrap an intent that will invoke **PhotoGalleryActivity** upon tapping on the notification

Create a notification with the given text, icon, intent to be fired upon tap (see *API doc for details*)

Delete notification from drawer when user taps on it

Post the notification

Running the App



Summary: Background Threads

- There are several mechanisms for executing work in background threads (i.e. other threads than the app's UI thread):
 - Android provides one background thread for “light”, short-lived, infrequent tasks tied to a particular activity (so called **AsyncTasks**) (*→ following slides; Android textbook, Ch. 23*)
 - All **AsyncTasks** are executed sequentially in the app's same background thread
 - (Dis)advantage: You cannot/need not manage the tasks' threading mechanics
 - You can create your own background threads for “heavy”, long-running tasks that are tied to a particular activity (so called **HandlerThreads**) (*→ Android textbook, Ch. 24*)
 - Each **HandlerThread** is executed in a background thread of its own, controlled by an activity
 - (Dis)advantage: You can/need to manage the threading mechanics yourself
 - You can create your own background threads for any tasks that are executed independently from activities (so called **IntentServices**) (*→ end of this class; Android textbook, Ch. 26*)
 - Each **IntentService** is executed in a thread of its own, started via an intent
 - Advantages: Completely independent from an app's UI, can be scheduled

Outlook: There's Much More...

*See the Android textbook and
developer.android.com/guide/*

- Invoking activities of other apps with implicit intents
- Notifying many apps at once with broadcast intents
- Various UI widgets (lists; paged, tabbed, modal dialogs; toolbars, menus etc.)
- Using the camera
- Using location information
- Using the accelerometer, magnetometer etc.
- Multimedia features (animations, sound, themes, styles etc.)
- Using the web browser, maps etc.
- Direct access to touch events and the drawing canvas
- Build apps for Android Wear, TV, automotive and other devices
- ...plus everything you can do with the regular Java SE API

Have fun!



In-Class Quiz 8: Threading



- Note whether the following properties apply to **(A)syncTasks, (I)ntentServices, (H)andlerThreads**
(note that some properties may apply to several threading mechanisms):
 - a) Executed in app's background thread
 - b) Executed in background thread of its own
 - c) Scheduling controlled by Android OS
 - d) Scheduling controlled by developer
 - e) Started via intent
 - f) Started via method invocation
 - g) Independent of particular activities
 - h) Tied to particular activity