Getting Started with Fresco

Getting Started with Fresco

This Guide will walk you through the steps needed to start using Fresco in your app, including loading your first image.

1. Update Gradle configuration

Edit your `build.gradle` file. You must add the following line to the `dependencies` section:

```
dependencies {
  // your app's other dependencies
  implementation 'com.facebook.fresco:fresco:1.13.0'
}
```

The following optional modules may also be added, depending on the needs of your app.

```
 dependencies {

   // For animated GIF support
   implementation 'com.facebook.fresco:animated-gif:1.13.0'

   // For WebP support, including animated WebP
   implementation 'com.facebook.fresco:animated-webp:1.13.0'
   implementation 'com.facebook.fresco:webpsupport:1.13.0'

   // For WebP support, without animations
   implementation 'com.facebook.fresco:webpsupport:1.13.0'

   // Provide the Android support library (you might already have this or a similar dependency)
   implementation 'com.android.support:support-core-utils:24.2.1'
}
```

2. Initialize Fresco & Declare Permissions

Fresco needs to be initialized. You should only do this 1 time, so placing the initialization in your Application is a good idea. An example for this would be:

```
[MyApplication.java]
public class MyApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        Fresco.initialize(this);
    }
}
```

NOTE: Remember to also declare you Application class in the `AndroidManifest.xml` as well as add the required permissions. In most cases you will need the INTERNET permission.

```
   <manifest
     ...
     >
     <uses-permission android:name="android.permission.INTERNET" />
     <application
       ...
       android:label="@string/app_name"
       android:name=".MyApplication"
       >
       ...
    </application>
    ...
  </manifest>
```

3. Create a Layout

In your layout XML, add a custom namespace to the top-level element. This is needed to access the custom `fresco:` attributes which allows you to control how the image is loaded and displayed.

```
<!-- Any valid element will do here -->
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:fresco="http://schemas.android.com/apk/res-auto"
    android:layout_height="match_parent"
    android:layout_width="match_parent"
    >
```

Then add the `SimpleDraweeView` to the layout:

```
<com.facebook.drawee.view.SimpleDraweeView
    android:id="@+id/my_image_view"
    android:layout_width="130dp"
    android:layout_height="130dp"
    fresco:placeholderImage="@drawable/my_drawable"
    />
```

To show an image, you need only do this:

```
Uri uri = Uri.parse("https://raw.githubusercontent.com/facebook/fresco/master/docs/static/logo.png");
SimpleDraweeView draweeView = (SimpleDraweeView) findViewById(R.id.my_image_view);
draweeView.setImageURI(uri);
```

and Fresco does the rest.

The placeholder is shown until the image is ready. The image will be downloaded, cached, displayed, and cleared from memory when your view goes off-screen.

Shipping Your App with Fresco

Shipping Your App with Fresco

Fresco's large size may seem intimidating, but it need not leave you with a large app. We strongly recommend use of the ProGuard tool as well as building split APKs to keep your app small.

ProGuard

Since Fresco 1.9.0 a ProGuard configuration file is included in Fresco itself which is automatically applied if you enable ProGuard for your app. To enable ProGuard, modify your `build.gradle` file to include the lines contained in the `release` section below.

```
android {
  buildTypes {
    release {
      minifyEnabled true
      proguardFiles getDefaultProguardFile('proguard-android.txt')
    }
  }
}
```

Build Multiple APKs

Fresco is written mostly in Java, but there is some C++ as well. C++ code has to be compiled for each of the CPU types (called "ABIs") Android can run on. Currently, Fresco supports five ABIs.

1. `armeabiv-v7a`: Version 7 or higher of the ARM processor. Most Android phones released from 2011-15 are using this.
2. `arm64-v8a`: 64-bit ARM processors. Found on new devices, like the Samsung Galaxy S6.
3. `x86`: Mostly used by tablets, and by emulators.
4. `x86_64`: Used by 64-bit tablets.

Fresco's binary download has copies of native `.so` files for all five platforms. You can reduce the size of your app considerably by creating separate APKs for each processor type.

If your app does not support Android 2.3 (Gingerbread) you will not need the `armeabi` flavor.

To enable multiple APKs, add the `splits` section below to the `android` section of your `build.gradle` file.

```
 android {
   // rest of your app's logic
   splits {
     abi {
         enable true
         reset()
         include 'x86', 'x86_64', 'arm64-v8a', 'armeabi-v7a'
         universalApk false
     }
   }
}
```

See the Android publishing documentation for more details on how splits work.

Using SimpleDraweeView

Using SimpleDraweeView

When using Fresco, you will use `SimpleDraweeView` to display images. These can be used in XML layouts. The simplest usage example of `SimpleDraweeView` is:

```
<com.facebook.drawee.view.SimpleDraweeView
  android:id="@+id/my_image_view"
  android:layout_width="20dp"
  android:layout_height="20dp"
  />
```

NOTE: `SimpleDraweeView` does not support `wrap_content` for `layout_width` or `layout_height` attributes. More information can be found here. The only exception to this is when you are setting an aspect ratio, like so:

```
<com.facebook.drawee.view.SimpleDraweeView
    android:id="@+id/my_image_view"
    android:layout_width="20dp"
    android:layout_height="wrap_content"
    fresco:viewAspectRatio="1.33"
    />
```

Loading an image

The easiest way to load an image into a `SimpleDraweeView` is to call `setImageURI`:

```
mSimpleDraweeView.setImageURI(uri);
```

That's it, you are now displaying images with Fresco!

Advanced XML attributes

`SimpleDraweeView`, despite its name, supports a great deal of customization through XML attributes. The example below presents all of them:

```
<com.facebook.drawee.view.SimpleDraweeView
  android:id="@+id/my_image_view"
  android:layout_width="20dp"
  android:layout_height="20dp"
  fresco:fadeDuration="300"
  fresco:actualImageScaleType="focusCrop"
  fresco:placeholderImage="@color/wait_color"
  fresco:placeholderImageScaleType="fitCenter"
  fresco:failureImage="@drawable/error"
  fresco:failureImageScaleType="centerInside"
  fresco:retryImage="@drawable/retrying"
  fresco:retryImageScaleType="centerCrop"
  fresco:progressBarImage="@drawable/progress_bar"
  fresco:progressBarImageScaleType="centerInside"
  fresco:progressBarAutoRotateInterval="1000"
  fresco:backgroundImage="@color/blue"
  fresco:overlayImage="@drawable/watermark"
  fresco:pressedStateOverlayImage="@color/red"
  fresco:roundAsCircle="false"
  fresco:roundedCornerRadius="1dp"
  fresco:roundTopLeft="true"
  fresco:roundTopRight="false"
  fresco:roundBottomLeft="false"
  fresco:roundBottomRight="true"
  fresco:roundTopStart="false"
  fresco:roundTopEnd="false"
  fresco:roundBottomStart="false"
  fresco:roundBottomEnd="false"
  fresco:roundWithOverlayColor="@color/corner_color"
  fresco:roundingBorderWidth="2dp"
  fresco:roundingBorderColor="@color/border_color"
  />
```

Customizing from code

Although it's generally recommended to set these options in XML, all of the attributes above can also be set from code. In order to do this, you will need to create a `DraweeHierarchy` before setting the image URI:

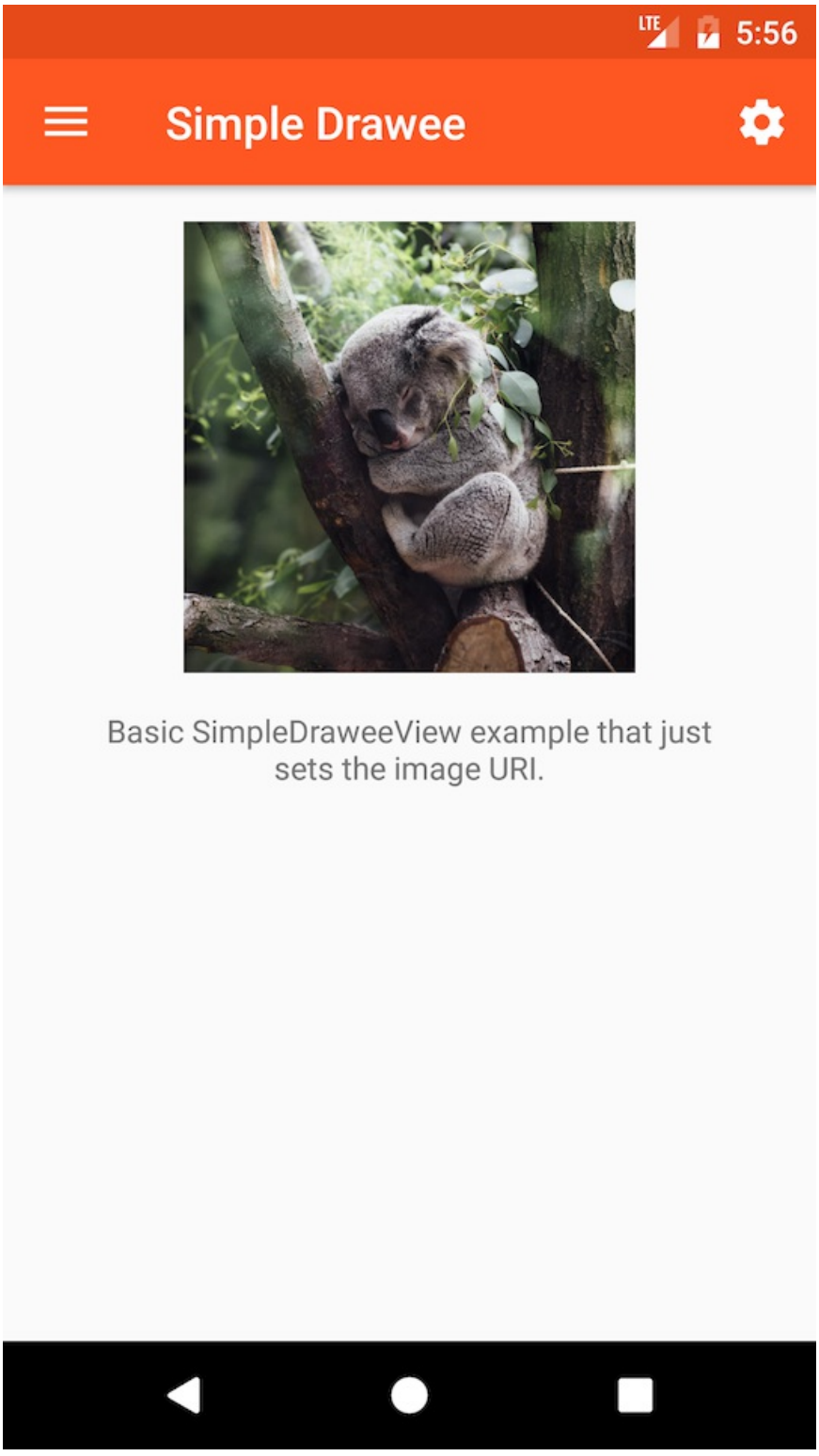
```
GenericDraweeHierarchy hierarchy =
    GenericDraweeHierarchyBuilder.newInstance(getResources())
        .setActualImageColorFilter(colorFilter)
        .setActualImageFocusPoint(focusPoint)
        .setActualImageScaleType(scaleType)
        .setBackground(background)
        .setDesiredAspectRatio(desiredAspectRatio)
        .setFadeDuration(fadeDuration)
        .setFailureImage(failureImage)
        .setFailureImageScaleType(scaleType)
        .setOverlays(overlays)
        .setPlaceholderImage(placeholderImage)
        .setPlaceholderImageScaleType(scaleType)
        .setPressedStateOverlay(overlay)
        .setProgressBarImage(progressBarImage)
        .setProgressBarImageScaleType(scaleType)
        .setRetryImage(retryImage)
        .setRetryImageScaleType(scaleType)
        .setRoundingParams(roundingParams)
        .build();
mSimpleDraweeView.setHierarchy(hierarchy);
mSimpleDraweeView.setImageURI(uri);
```

NOTE: some of these options can be set on an existing hierarchy without having to build a new one. To do this, simply get the hierarchy from a `SimpleDraweeView` and call any of the setter methods on it, e.g.:

```
mSimpleDraweeView.getHierarchy().setPlaceHolderImage(placeholderImage);
```

Full Sample

For a full sample see the `DraweeSimpleFragment` in the showcase app: DraweeSimpleFragment.java


LTE
5:56
Simple Drawee
Basic SimpleDraweeView example that just sets the image URI.

Rounded Corners and Circles

Rounded Corners and Circles

Not every image is a rectangle. Apps frequently need images that appear with softer, rounded corners, or as circles. Drawee supports a variety of scenarios, all without the memory overhead of copying bitmaps.

What

Images can be rounded in two shapes:

1. As a circle - set `roundAsCircle` to true.
2. As a rectangle, but with rounded corners. Set `roundedCornerRadius` to some value.

Rectangles support having each of the four corners have a different radius, but this must be specified in Java code rather than XML.

How

Images can be rounded with two different methods:

1. `BITMAP_ONLY` - Uses a bitmap shader to draw the bitmap with rounded corners. This is the default rounding method. It doesn't support animations, and it does **not** support any scale types other than `centerCrop` (the default), `focusCrop` and `fit_xy`. If you use this rounding method with other scale types, such as `center`, you won't get an Exception but the image might look wrong (e.g. repeated edges due to how Android shaders work), especially in cases the source image is smaller than the view. See the Caveats section below.
2. `OVERLAY_COLOR` - Draws rounded corners by overlaying a solid color, specified by the caller. The Drawee's background should be static and of the same solid color. Use `roundWithOverlayColor` in XML, or `setOverlayColor` in code to use this rounding method.

In XML

The `SimpleDraweeView` class will forward several attributes over to `RoundingParams`:

```
<com.facebook.drawee.view.SimpleDraweeView
  ...
  fresco:roundedCornerRadius="5dp"
  fresco:roundBottomStart="false"
  fresco:roundBottomEnd="false"
  fresco:roundWithOverlayColor="@color/blue"
  fresco:roundingBorderWidth="1dp"
  fresco:roundingBorderColor="@color/red"
>
```

In code

When constructing a hierarchy, you can pass an instance of RoundingParams to your `GenericDraweeHierarchyBuilder`:

```
int overlayColor = getResources().getColor(R.color.green);
RoundingParams roundingParams = RoundingParams.fromCornersRadius(7f);
mSimpleDraweeView.setHierarchy(new GenericDraweeHierarchyBuilder(getResources())
        .setRoundingParams(roundingParams)
        .build());
```

You can also change all of the rounding parameters after the hierarchy has been built:

```
1 int color = getResources().getColor(R.color.red);
2 RoundingParams roundingParams = RoundingParams.fromCornersRadius(5f);
3 roundingParams.setBorder(color, 1.0f);
4 roundingParams.setRoundAsCircle(true);
5 mSimpleDraweeView.getHierarchy().setRoundingParams(roundingParams);
```

Caveats

There are some limitations when `BITMAP_ONLY` (the default) mode is used:

- Only images that resolve to `BitmapDrawable` or `ColorDrawable` can be rounded. Rounding `NinePatchDrawable`, `ShapeDrawable` and other such drawables is not supported (regardless whether they are specified in XML or programmatically).
- Animations are not rounded.
- Due to a limitation of Android's `BitmapShader`, if the image doesn't fully cover the view, instead of drawing nothing, edges are repeated. One workaround is to use a different scale type (e.g. centerCrop) that ensures that the whole view is covered. Another workaround is to make the image file contain a 1px transparent border so that the transparent pixels get repeated. This is the best solution for PNG resource images.

If the limitations of the `BITMAP_ONLY` mode affect your images, see if the `OVERLAY_COLOR` mode works for you. The `OVERLAY_COLOR` mode doesn't have the aforementioned limitations, but since it simulates rounded corners by overlaying a solid color over the image, this only looks good if the background under the view is static and of the same color.

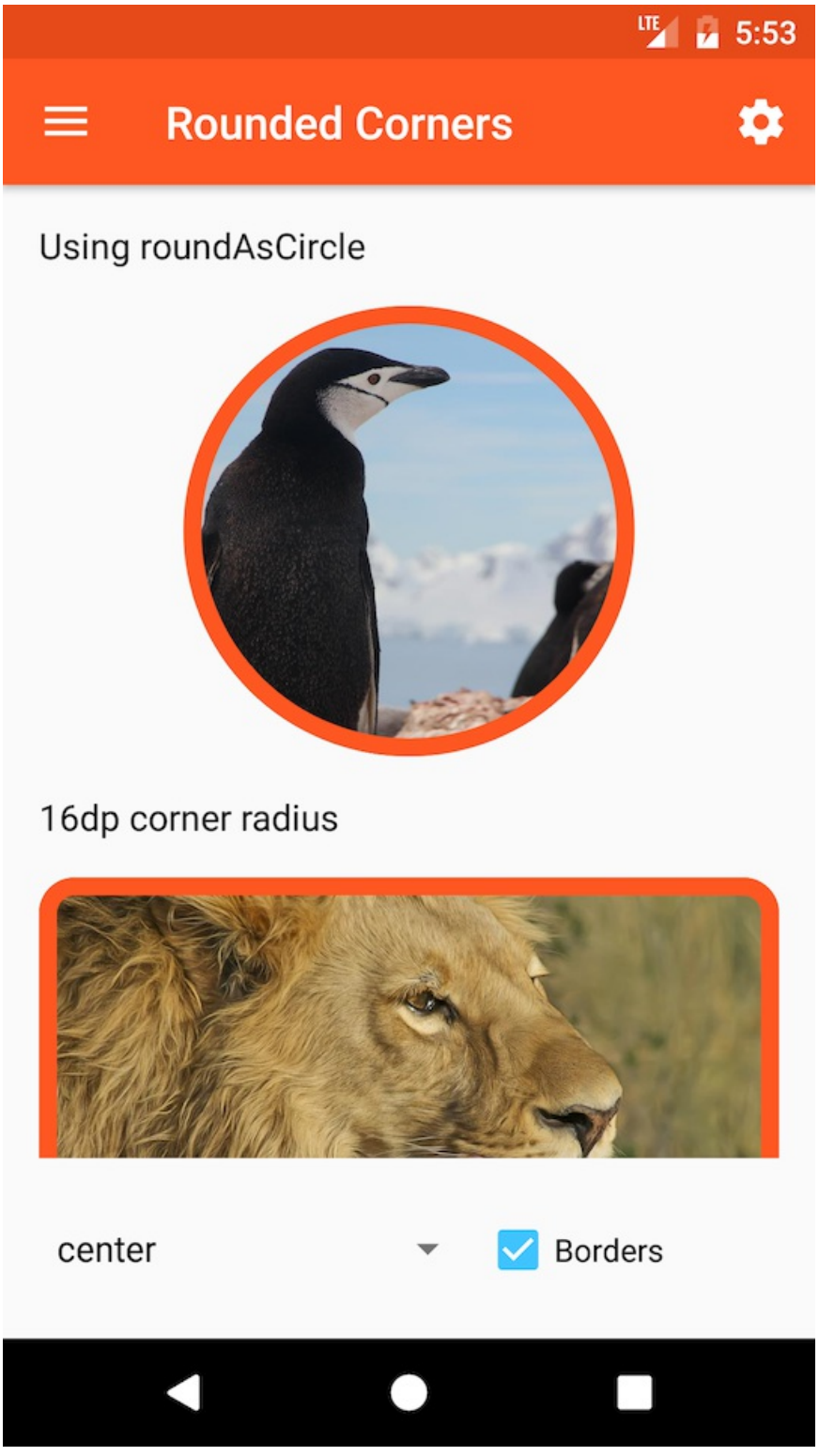
Drawee internally has an implementation for `CLIPPING` mode, but this mode has been disabled and not exposed as some `Canvas` implementation do not support path clipping. Furthermore, canvas clipping doesn't support antialiasing which makes the rounded edges very pixelated.

Finally, all of those issues could be avoided by using a temporary bitmap, but this imposes a significant memory overhead and has not been supported because of that.

As explained above, there is no really good solution for rounding corners on Android and one has to choose between the aforementioned trade-offs.

Full Sample

For a full sample see the `DraweeRoundedCornersFragment` in the showcase app: DraweeRoundedCornersFragment.java


5:53
Rounded Corners
Using roundAsCircle
16dp corner radius
center
Borders

Progress Bars

Progress Bars

The easiest way to set a progress bar in your application is to use the ProgressBarDrawable class when building a hierarchy:

```
1 .setProgressBarImage(new ProgressBarDrawable())
```

This shows the progress bar as a dark blue rectangle along the bottom of the Drawee.

Defining your own progress bar

If you wish to customize your own progress indicator, be aware that in order for it to accurately reflect progress while loading, it needs to override the Drawable.onLevelChange method:

```
1  class CustomProgressBar extends Drawable {
2     @Override
3     protected boolean onLevelChange(int level) {
4       // level is on a scale of 0-10,000
5       // where 10,000 means fully downloaded
6
7       // your app's logic to change the drawable's
8       // appearance here based on progress
9     }
10 }
```

Example

The Fresco showcase app has a DraweeHierarchyFragment that demonstrates using a progress bar drawable.

ScaleTypes

ScaleTypes

You can specify a different scale type for each of the different drawables in your Drawee.

Available Scale Types

Scale Type	Preserves Aspect Ratio	Always Fills Entire View	Performs Scaling	Explanation
center	✓			Center the image in the view, but perform no scaling.
centerCrop	✓	✓	✓	Scales the image so that both dimensions will be greater than or equal to the corresponding dimension of the parent. One of width or height will fit exactly. The image is centered within parent's bounds.
focusCrop	✓	✓	✓	Same as centerCrop, but based around a caller-specified focus point instead of the center.
centerInside	✓		✓	Downscales the image so that it fits entirely inside the parent. Unlike `fitCenter`, no upscaling will be performed. Aspect ratio is preserved. The image is centered within parent's bounds.
fitCenter	✓		✓	Scales the image so that it fits entirely inside the parent. One of width or height will fit exactly. Aspect ratio is preserved. The image is centered within the parent's bounds.
fitStart	✓		✓	Scales the image so that it fits entirely inside the parent. One of width or height will fit exactly. Aspect ratio is preserved. The image is aligned to the top-left corner of the parent.
fitEnd	✓		✓	Scales the image so that it fits entirely inside the parent. One of width or height will fit exactly. Aspect ratio is preserved. The image is aligned to the bottom-right corner of the parent.
fitXY		✓	✓	Scales width and height independently. The image will match the parent exactly. Aspect ratio is not preserved.
none	✓			Used for Android's tile mode.

These are mostly the same as those supported by the Android ImageView class. The one unsupported type is `matrix`. In its place, Fresco offers `focusCrop`, which will usually work better.

How to Set a Scale Type

ScaleTypes of actual, placeholder, retry, and failure images can all be set in XML, using attributes like `fresco:actualImageScaleType`. You can also set them in code using the GenericDraweeHierarchyBuilder class.

Even after your hierarchy is built, the actual image scale type can be modified on the fly using GenericDraweeHierarchy.

However, do **not** use the `android:scaleType` attribute, nor the `.setScaleType` method. These have no effect on Drawees.

Scale Type: "focusCrop"

Android, and Fresco, offer a `centerCrop` scale type, which will fill the entire viewing area while preserving the aspect ratio of the image, cropping as necessary.

This is very useful, but the trouble is the cropping doesn't always happen where you need it. If, for instance, you want to crop to someone's face in the bottom right corner of the image, `centerCrop` will do the wrong thing.

By specifying a focus point, you can say which part of the image should be centered in the view. If you specify the focus point to be at the top of the image, such as (0.5f, 0f), we guarantee that, no matter what, this point will be visible and centered in the view as much as possible.

Focus points are specified in a relative coordinate system. That is, (0f, 0f) is the top-left corner, and (1f, 1f) is the bottom-right corner. Relative coordinates allow focus points to be scale-invariant, which is highly useful.

A focus point of (0.5f, 0.5f) is equivalent to a scale type of `centerCrop`.

To use focus points, you must first set the right scale type in your XML:

```
1  fresco:actualImageScaleType="focusCrop"
```

In your Java code, you must programmatically set the correct focus point for your image:

```
PointF focusPoint = new PointF(0f, 0.5f);
mSimpleDraweeView
    .getHierarchy()
    .setActualImageFocusPoint(focusPoint);
```

ScaleType: “none”

If you are using Drawables that make use of Android’s tile mode, you need to use the `none` scale type for this to work correctly.

Scale Type: A Custom ScaleType

Sometimes you need to scale the image in a way that none of the existing scale types does. Drawee allows you to do that easily by implementing your own `ScalingUtils.ScaleType`. There is only one method in that interface, `getTransform`, which is supposed to compute the transformation matrix based on:

- parent bounds (rectangle where the image should be placed in the view’s coordinate system)
- child size (width and height of the actual bitmap)
- focus point (relative coordinates in the child’s coordinate system)

Of course, your class can contain any additional data you might need to compute the transformation.

Let’s look at an example. Assume the `parentBounds` are `(100, 150, 500, 450)`, and the child dimensions are `(420,210)`. Observe that the parent width is `500 - 100 = 400`, and the height is `450 - 150 = 300`. If we don’t do any transformation (i.e. we set the transformation to be the identity matrix), the image will be drawn in `(0, 0, 420, 210)`. But `ScaleTypeDrawable` has to respect the bounds imposed by the parent and will so clip the canvas to `(100, 150, 500, 450)`. That means that only the bottom-right part of the image will actually be visible: `(100, 150, 420, 210)`.

We can fix that by doing a translation by `(parentBounds.left, parentBounds.top)`, which is in this case `(100, 150)`. But now the right part of the image got clipped as the image is actually wider than the parent bounds! Image is now placed at `(100, 150, 500, 360)` in the view coordinates, or equivalently `(0, 0, 400, 210)` in the child coordinates. We lost `20` pixels on the right.

To avoid image to be clipped we can downscale it. Here we can scale by `400/420` which will make the image be of the size `(400,200)`. The image now fits exactly in the view horizontally, but it is not centered in it vertically.

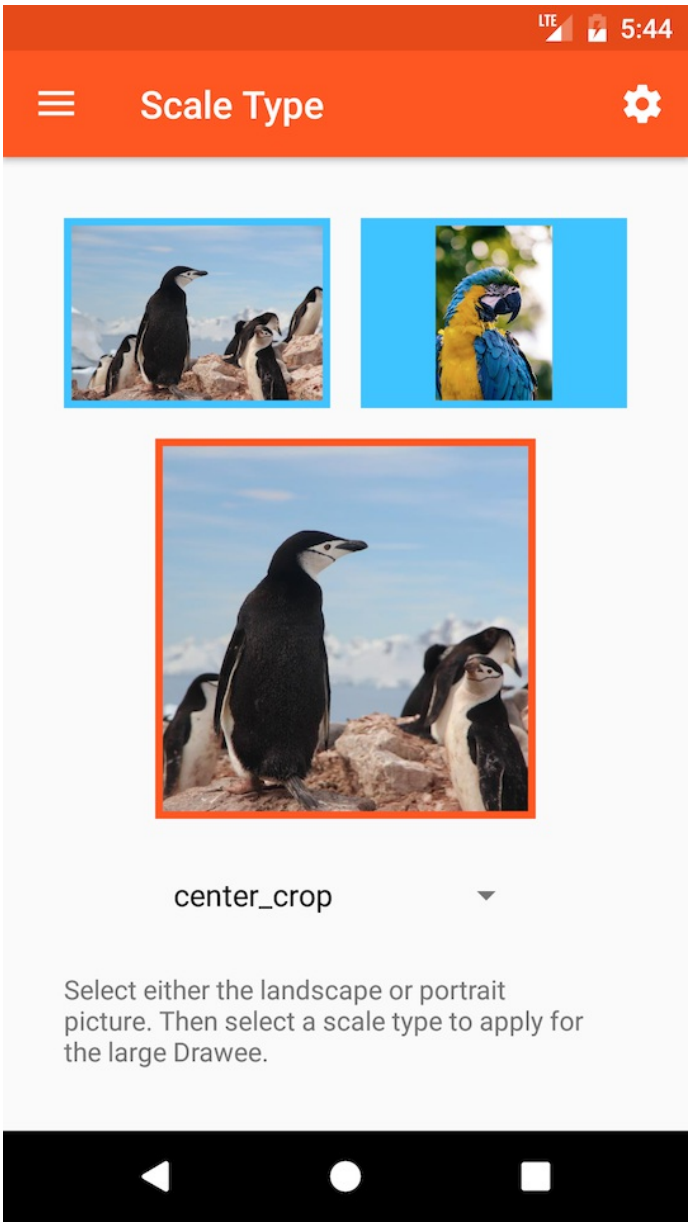
In order to center the image we need to translate it a bit more. We can see that the amount of empty space in the parent bounds is `400 - 400 = 0` horizontally, and `300 - 200 = 100` vertically. If we translate by half of this empty space, we will leave equal amount of empty space on each side, effectively making the image centered in the parent bounds.

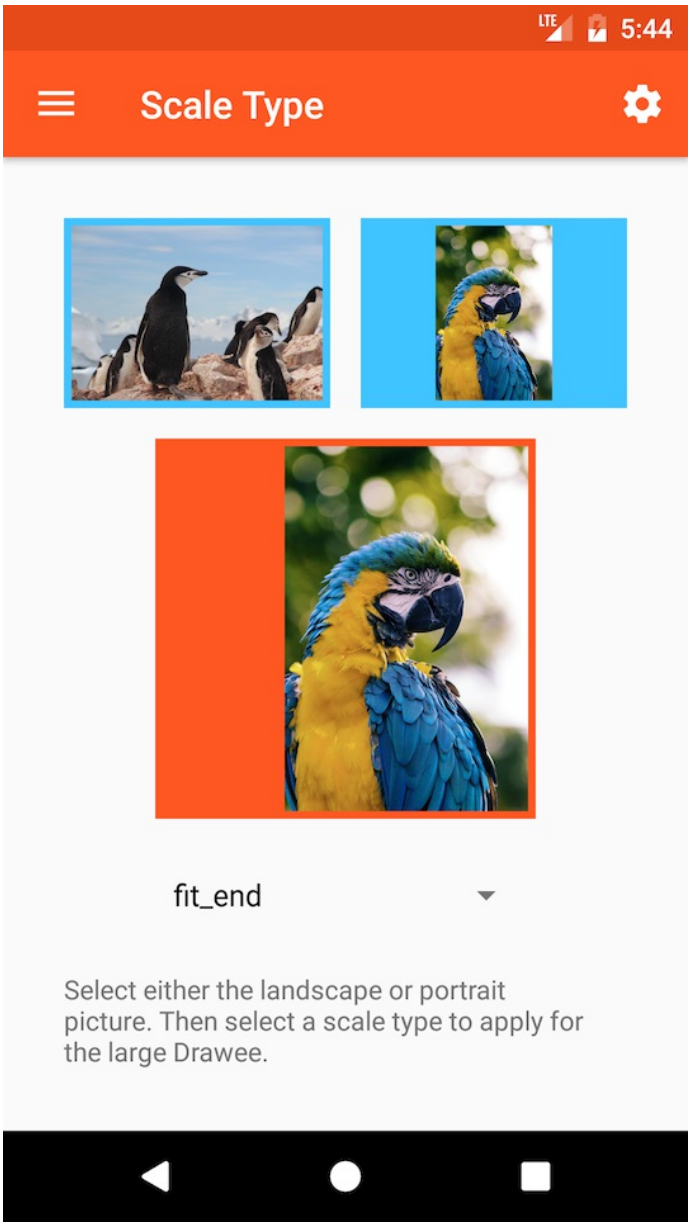
Congratulations! You just implemented the `FIT_CENTER` scale type:

```
  public class AbstractScaleType implements ScaleType {
    @Override
    public Matrix getTransform(Matrix outTransform, Rect parentRect, int childWidth, int childHeight, float focusX, float focusY) {
      // calculate scale; we take the smaller of the horizontal and vertical scale factor so that the image always fits
      final float scaleX = (float) parentRect.width() / (float) childWidth;
      final float scaleY = (float) parentRect.height() / (float) childHeight;
      final float scale = Math.min(scaleX, scaleY);

      // calculate translation; we offset by parent bounds, and by half of the empty space
      // note that the child dimensions need to be adjusted by the scale factor
      final float dx = parentRect.left + (parentRect.width() - childWidth * scale) * 0.5f;
      final float dy = parentRect.top + (parentRect.height() - childHeight * scale) * 0.5f;

      // finally, set and return the transform
      outTransform.setScale(scale, scale);
      outTransform.postTranslate((int) (dx + 0.5f), (int) (dy + 0.5f));
      return outTransform;
    }
  }
```

Full Sample

For a full sample see the `DraweeScaleTypeFragment` in the showcase app: DraweeScaleTypeFragment.java


5:44
Scale Type
center_crop
Select either the landscape or portrait picture. Then select a scale type to apply for the large Drawee.


5:44
Scale Type
fit_end
Select either the landscape or portrait picture. Then select a scale type to apply for the large Drawee.

Placeholder, failure and retry images

Placeholder, failure and retry images

When you're loading network images things can go wrong, take a long time, or some images might not even be available at all. We've seen how to display progress bars. On this page, we look at the other things that a `SimpleDraweeView` can display while the actual image is not available (yet, or at all). Note that all of these can have different scale types, which you can customize.

Placeholder Image

The placeholder image is displayed from before you've set a URI or a controller until it has finished loading (successfully or not).

XML

```
<com.facebook.drawee.view.SimpleDraweeView
  android:id="@+id/my_image_view"
  android:layout_width="20dp"
  android:layout_height="20dp"
  fresco:placeholderImage="@drawable/my_placeholder_drawable"
  />
```

Code

```
mSimpleDraweeView.getHierarchy().setPlaceholderImage(placeholderImage);
```

Failure Image

The failure image is displayed when a request has completed in error, either network-related (404, timeout) or image data-related (malformed image, unsupported format).

XML

```
<com.facebook.drawee.view.SimpleDraweeView
  android:id="@+id/my_image_view"
  android:layout_width="20dp"
  android:layout_height="20dp"
  fresco:failureImage="@drawable/my_failure_drawable"
  />
```

Code

```
mSimpleDraweeView.getHierarchy().setFailureImage(failureImage);
```

Retry Image

The retry image appears instead of the failure image. When the user taps on it, the request is retried up to four times, before the failure image is displayed. In order for the retry image to work, you need to enable support for it in your controller, which means setting up your image request like so:

```
mSimpleDraweeView.setController(
    Fresco.newDraweeControllerBuilder()
        .setTapToRetryEnabled(true)
        .setUri(uri)
        .build());
```

XML

```
<com.facebook.drawee.view.SimpleDraweeView
  android:id="@+id/my_image_view"
  android:layout_width="20dp"
  android:layout_height="20dp"
  fresco:failureImage="@drawable/my_failure_drawable"
  />
```

Code

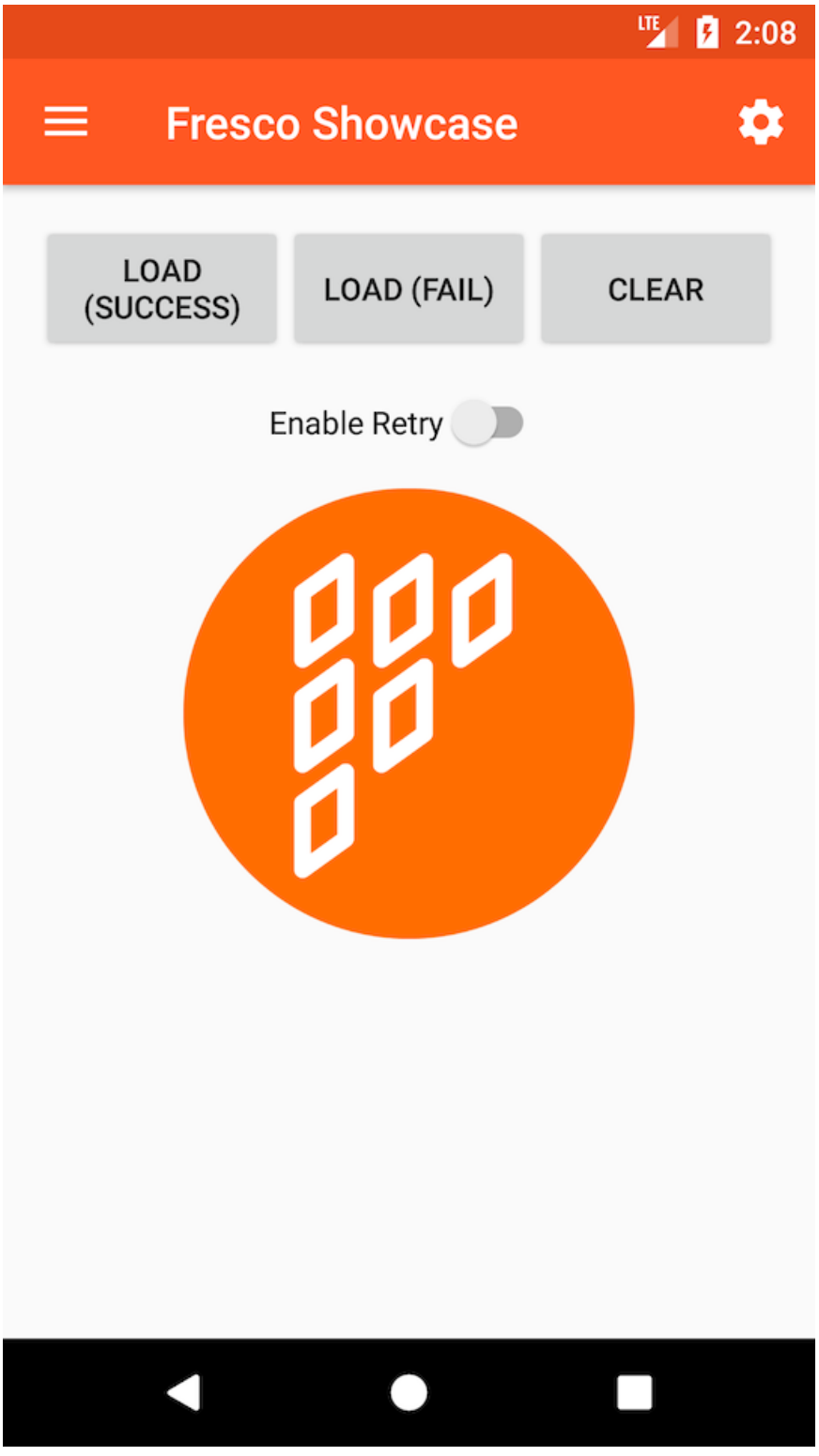
```
simpleDraweeView.getHierarchy().setRetryImage(retryImage);
```

Further Reading

Placeholder, failure and retry images are drawee *branches*. There are others than what is presented on this page, though these are the most commonly used ones. To read about all of the branches and how they work, check out drawee branches.

Example

The Fresco showcase app has a DraweeHierarchyFragment that demonstrates using placeholder, failure and retry images.


2:08
Fresco Showcase
LOAD (SUCCESS)
LOAD (FAIL)
CLEAR
Enable Retry

Rotation

Rotation

You can rotate images by specifying a rotation angle in the image request, like so:

```
final ImageRequest imageRequest = ImageRequestBuilder.newBuilderWithSource(uri)
    .setRotationOptions(RotationOptions.forceRotation(RotationOptions.ROTATE_90))
    .build();
mSimpleDraweeView.setController(
    Fresco.newDraweeControllerBuilder()
        .setImageRequest(imageRequest)
        .build());
```

Auto-rotation

JPEG files sometimes store orientation information in the image metadata. If you want images to be automatically rotated to match the device's orientation, you can do so in the image request:

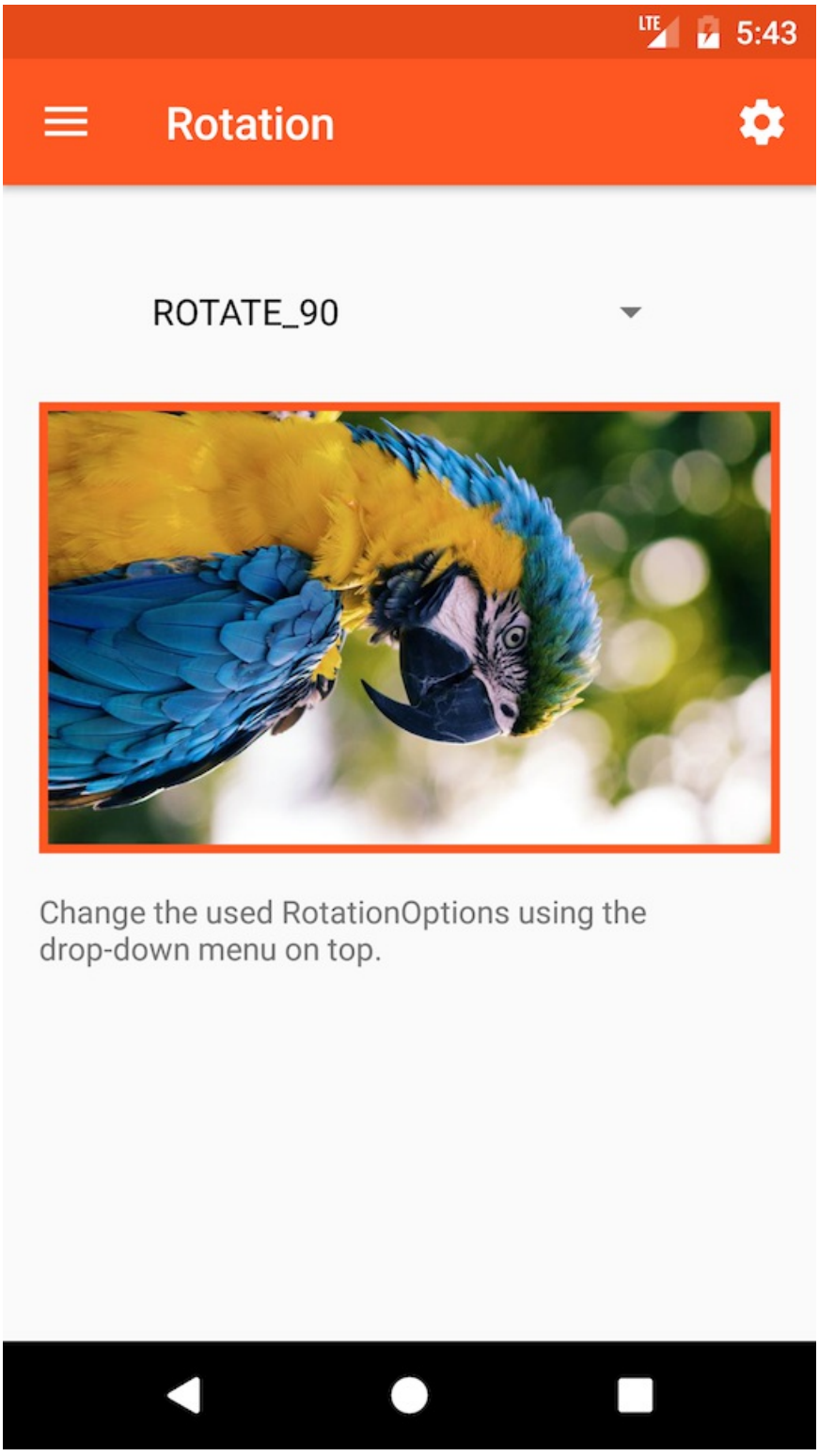
```
final ImageRequest imageRequest = ImageRequestBuilder.newBuilderWithSource(uri)
    .setRotationOptions(RotationOptions.autoRotate())
    .build();
mSimpleDraweeView.setController(
    Fresco.newDraweeControllerBuilder()
        .setImageRequest(imageRequest)
        .build());
```

Combining rotations

If you're loading a JPEG file that has rotation information in its EXIF data, calling `forceRotation` will **add** to the default rotation of the image. For example, if the EXIF header specifies 90 degrees, and you call `forceRotation(ROTATE_90)`, the raw image will be rotated 180 degrees.

Examples

The Fresco showcase app has a DraweeRotationFragment that demonstrates the various rotation settings. You can use it for example with the sample images from here.


5:43
Rotation
ROTATE_90
Change the used RotationOptions using the drop-down menu on top.

Resizing

Resizing

We use the following terminology for this section:

- **Scaling** is a canvas operation and is usually hardware accelerated. The bitmap itself is always the same size. It just gets drawn downscaled or upscaled. See ScaleTypes.
- **Resizing** is a pipeline operation executed in software. This changes the encoded image in memory before it is being decoded. The decoded bitmap will be smaller than the original image.
- **Downsampling** is also a pipeline operation implemented in software. Rather than creating a new encoded image, it simply decodes only a subset of the pixels, resulting in a smaller output bitmap.

Resizing

Resizing does not modify the original file, it just resizes an encoded image in memory, prior to being decoded.

To resize pass a `ResizeOptions` object when constructing an `ImageRequest`:

```
1 ImageRequest request = ImageRequestBuilder.newBuilderWithSource(uri)
2     .setResizeOptions(new ResizeOptions(50, 50))
3     .build();
4 mSimpleDraweeView.setController(
5     Fresco.newDraweeControllerBuilder()
6         .setOldController(mSimpleDraweeView.getController())
7         .setImageRequest(request)
8         .build());
```

Resizing has some limitations:

- it only supports JPEG files
- the actual resize is carried out to the nearest 1/8 of the original size
- it cannot make your image bigger, only smaller (not a real limitation though)

Downsampling

Downsampling is an experimental feature added recently to Fresco. To use it, you must explicitly enable it when configuring the image pipeline:

```
1     .setDownsampleEnabled(true)
```

If this option is on, the image pipeline will downsample your images instead of resizing them. You must still call `setResizeOptions` for each image request as above.

Downsampling is generally faster than resizing, since it is part of the decode step, rather than a separate step of its own. It also supports PNG and WebP (except animated) images as well as JPEG.

The trade-off right now is that, on Android 4.4 (KitKat) it uses more memory than resizing, while the decode is taking place. This should only be an issue for apps decoding a large number of images simultaneously. We hope to find a solution for this and make it the default in a future release.

Which should you use and when?

If the image is **not** much bigger than the view, then only scaling should be done. It's faster, easier to code, and results in a higher quality output. Of course, images smaller than the view are subset of those **not** much bigger than the view. Therefore, if you need to upscale the image, this should too be done by scaling, and not by resizing. That way memory won't be wasted on a larger bitmap that does not provide any better quality. However, for images much bigger than the view, such as **local camera images**, resizing in addition to scaling is highly recommended.

As for what exactly "much bigger" means, as a rule of thumb if the image is more than 2 times bigger than the view (in total

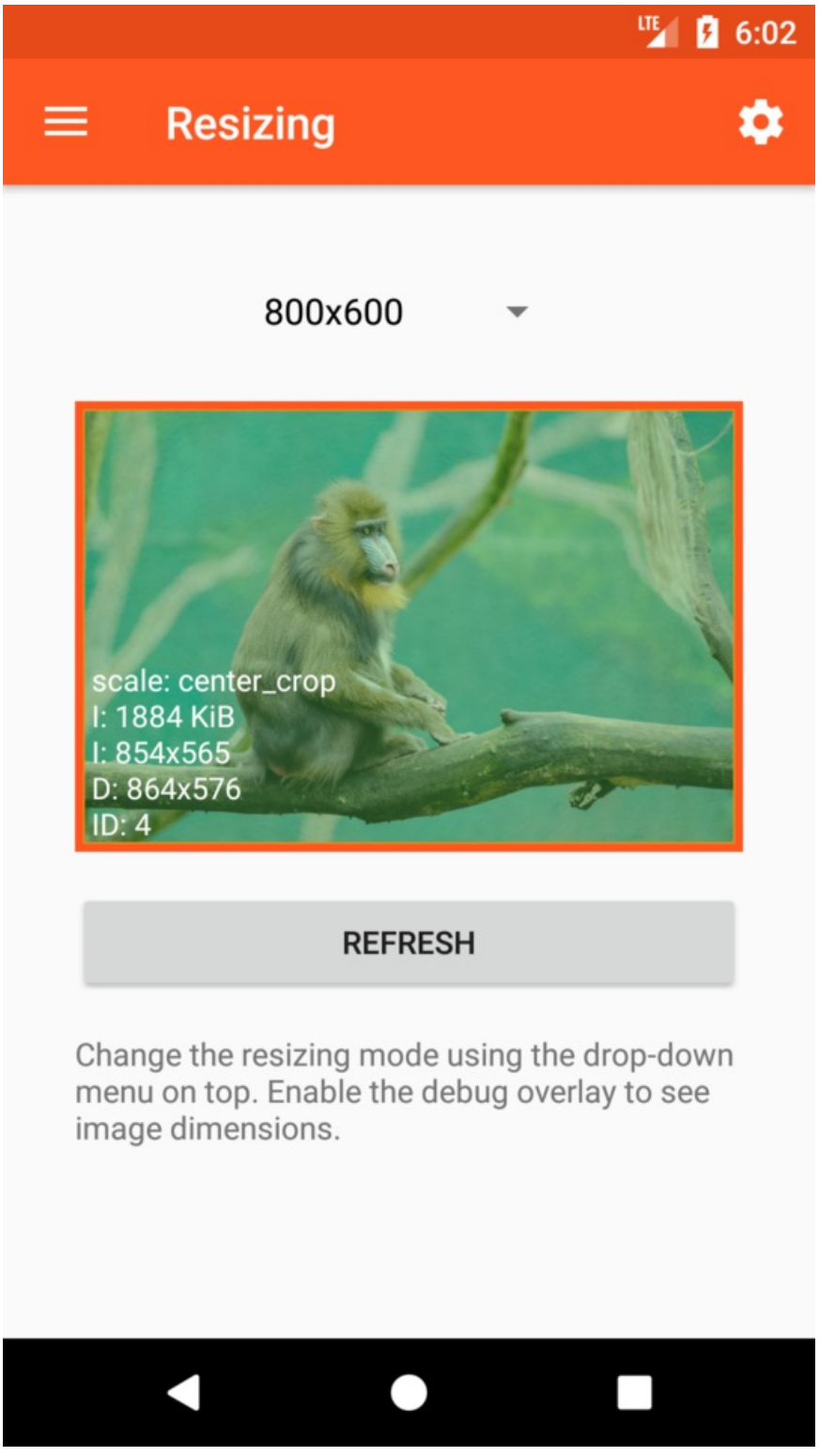
number of pixels, i.e. width*height), you should resize it. This almost always applies for local images taken by camera. For example, a device with the screen size of 1080 x 1920 pixels (roughly 2MP) and a camera of 16MP produces images 8 times bigger than the display. Without any doubt resizing in such cases is always best to be done.

For network images, try to download an image as close as possible to the size you will be displaying. By downloading images of inappropriate size you are wasting the user's time and data.

If the image is bigger than the view, by not resizing it the memory gets wasted. However, there is also a performance trade-off to be considered. Clearly, resizing imposes additional CPU cost on its own. But, by not resizing images bigger than the view, more bytes need to be transferred to the GPU, and images get evicted from the bitmap cache more often resulting in more decodes. In other words, not resizing when you should also imposes additional CPU cost. Therefore, there is no silver bullet and depending on the device characteristics there is a threshold point after which it becomes more performant to go with resize than without it.

Example

The Fresco showcase app has a ImagePipelineResizingFragment that demonstrates using placeholder, failure and retry images.


6:02
Resizing
800x600
scale: center_crop
I: 1884 KiB
I: 854x565
D: 864x576
ID: 4
REFRESH
Change the resizing mode using the drop-down menu on top. Enable the debug overlay to see image dimensions.

Supported URIs

Supported URIs

Fresco supports images in a variety of locations. Fresco does **not** accept relative URIs. All URIs must be absolute and must include the scheme.

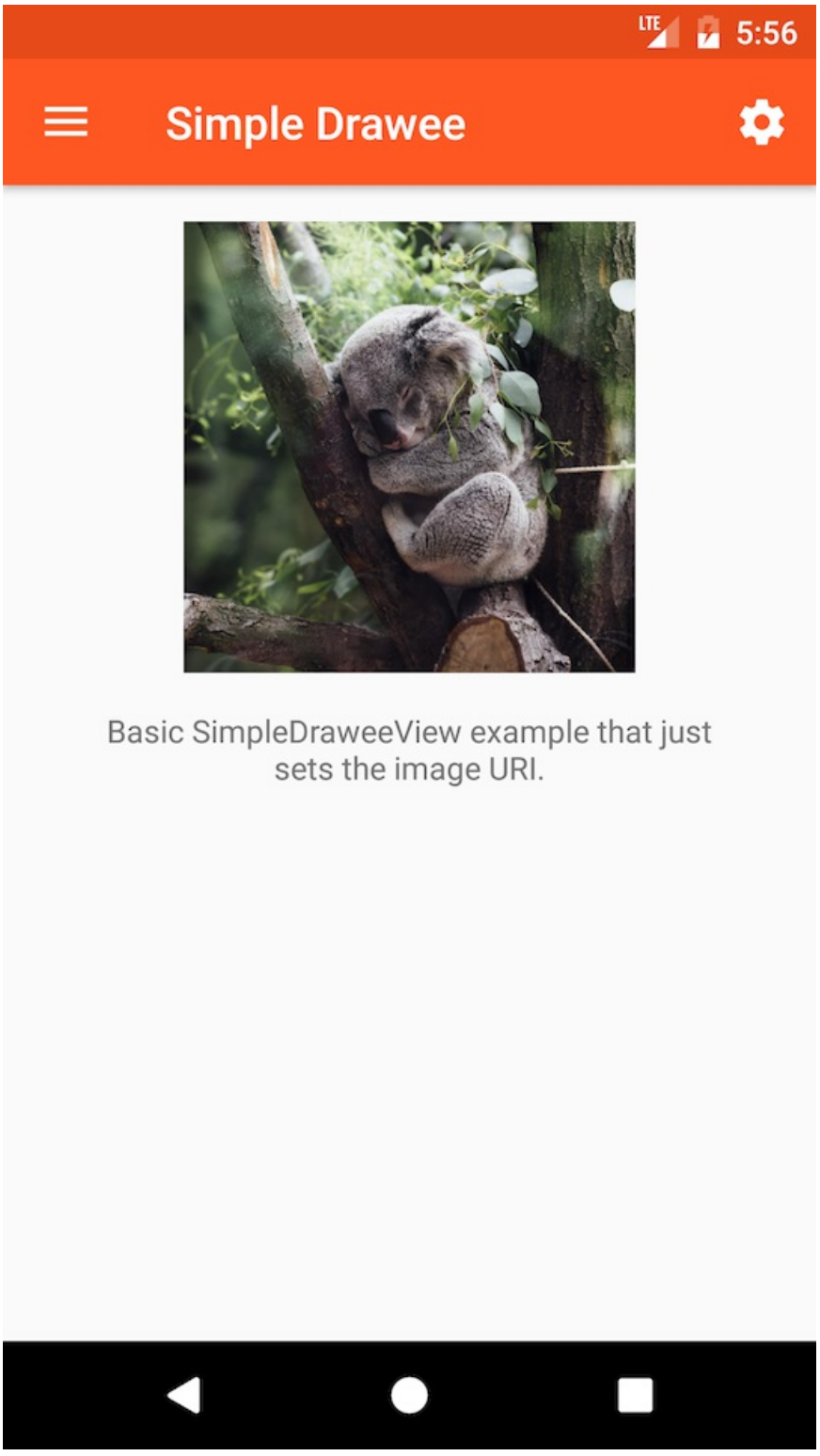
These are the URI schemes accepted:

Type	Scheme	Fetch method used
File on network	`http://`, `https://`	`HttpURLConnection` or network layer
File on device	`file://`	`FileInputStream`
Content provider	`content://`	`ContentResolver`
Asset in app	`asset://`	`AssetManager`
Resource in app	`res://` as in `res:///12345`	`Resources.openRawResource`
Data in URI	`data:mime/type;base64,`	Following data URI spec (UTF-8 only)

Note: Only image resources can be used with the image pipeline (e.g. a PNG image). Other resource types such as Strings or XML Drawables make no sense in the context of the image pipeline and so cannot be supported by definition. One potentially confusing case is drawable declared in XML (e.g. ShapeDrawable). Important thing to note is that this is **not** an image. If you want to display an XML drawable as the main image, then set it as a placeholder and use the `null` uri.

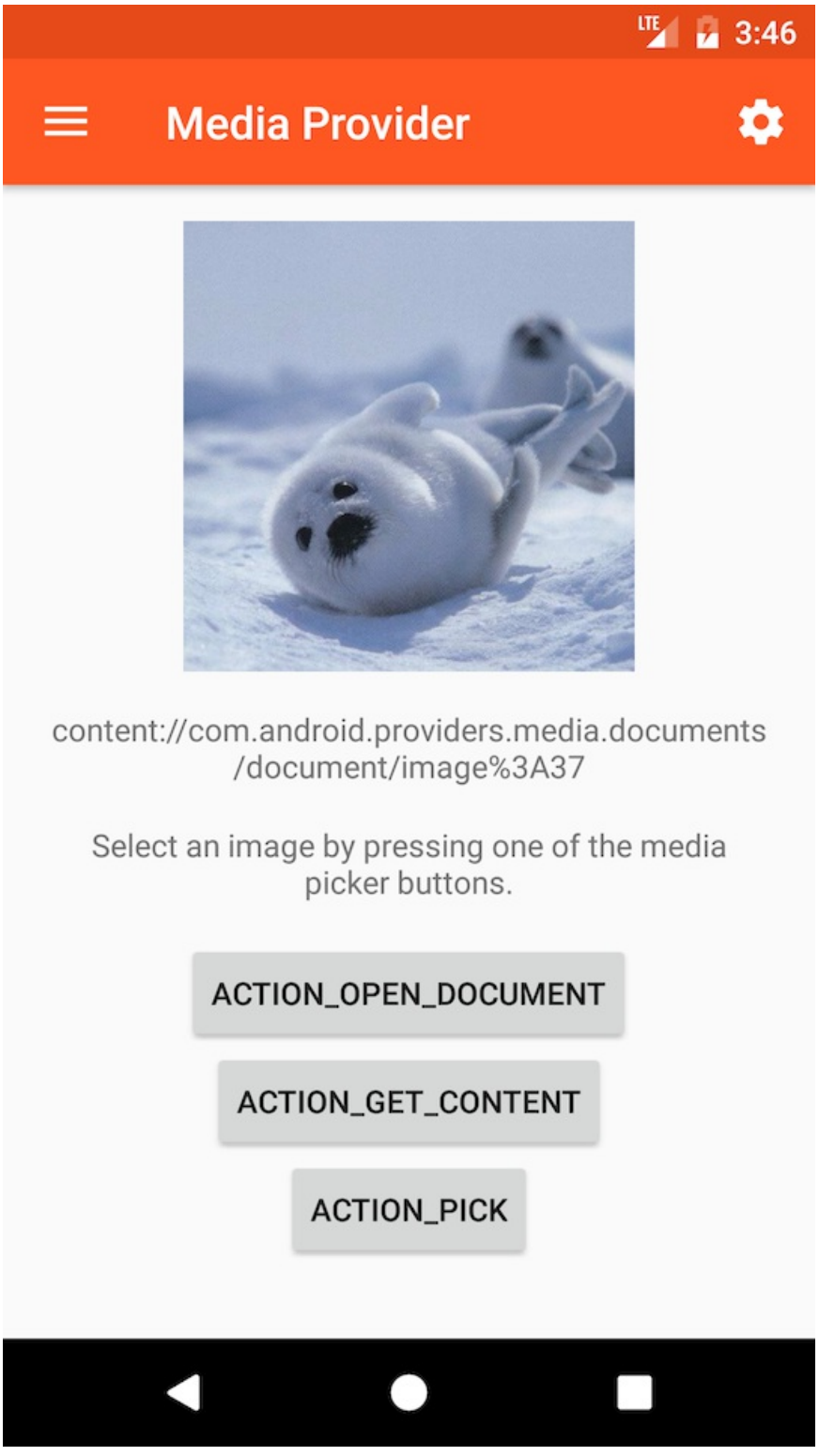
Sample: Loading an URI

For a sample that just loads an URI see the `DraweeSimpleFragment` in the showcase app: DraweeSimpleFragment.java



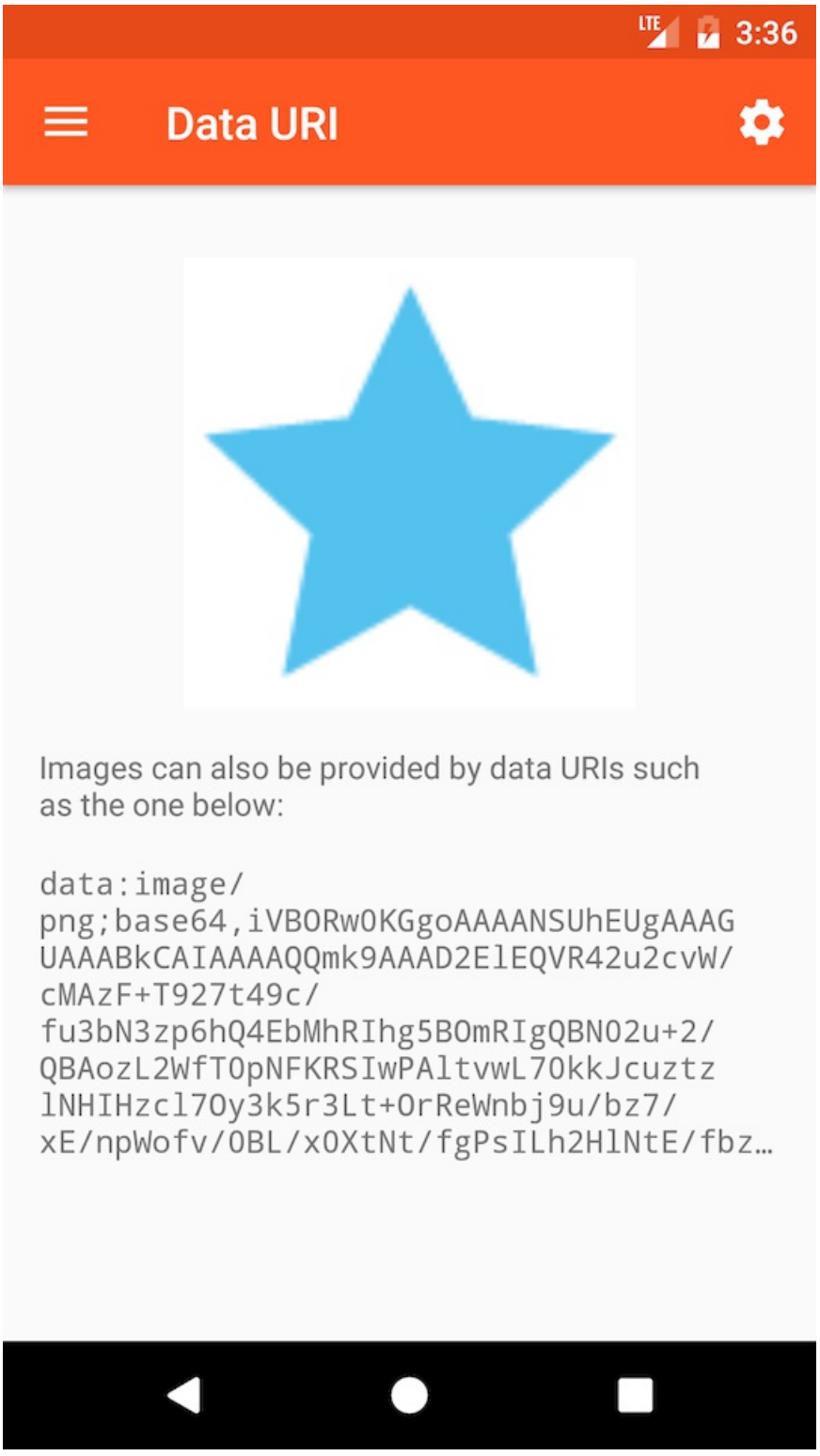
Sample: Loading a Local File

For a sample on how to correctly load user-selected files (e.g. using the `content://` URI) see the `DraweeMediaPickerFragment` in the showcase app: DraweeMediaPickerFragment.java



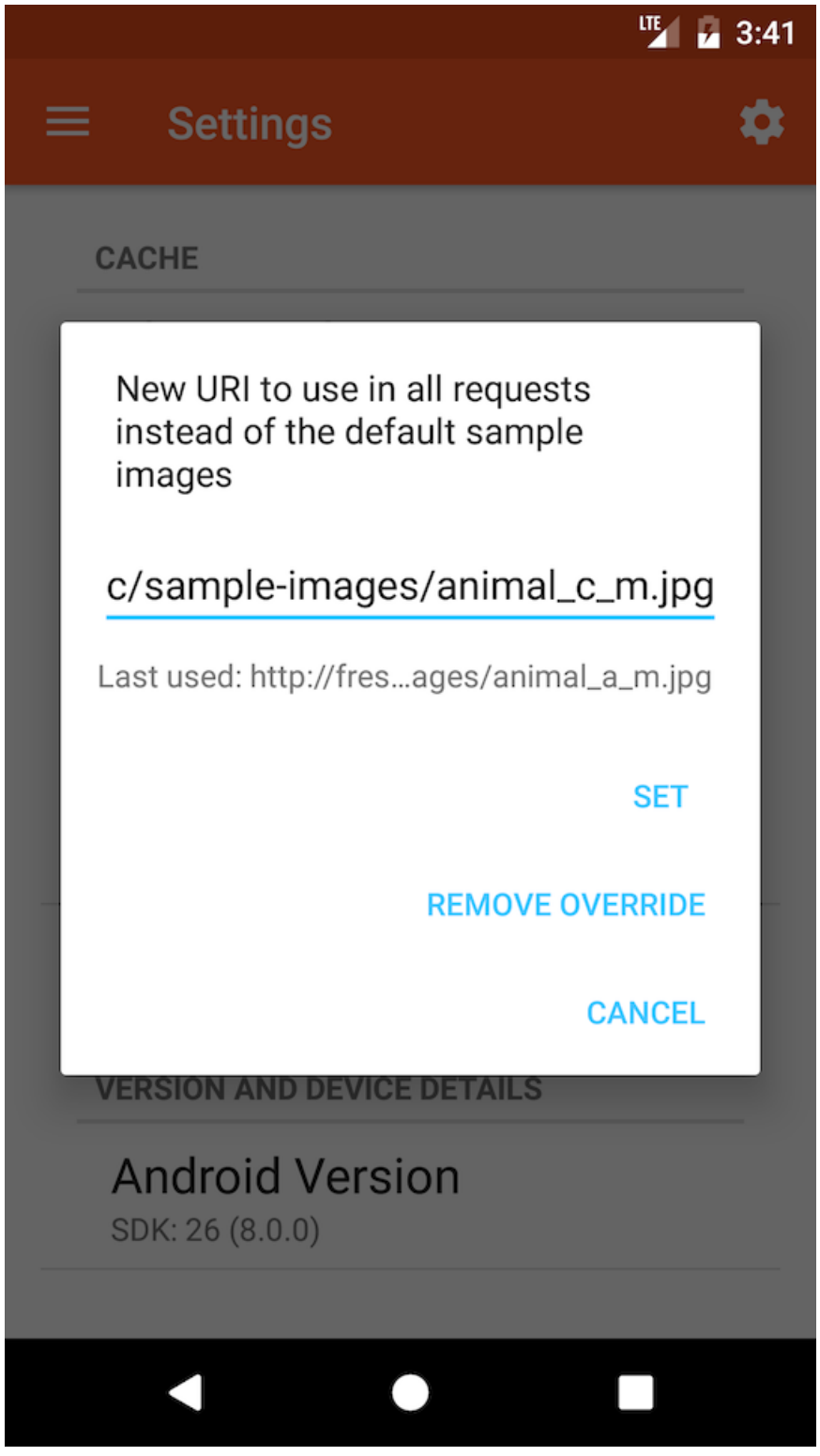
Sample: Loading a Data URI

The Fresco showcase app has a ImageFormatDataUriFragment that demonstrates using placeholder, failure and retry images.



More

Tip: You can override the displayed image URI in many samples in the showcase app by using the *URI Override* option in the global settings:


3:41
Settings
CACHE
New URI to use in all requests instead of the default sample images
c/sample-images/animal_c_m.jpg
Last used: http://fres…ages/animal_a_m.jpg
SET
REMOVE OVERRIDE
CANCEL
VERSION AND DEVICE DETAILS
Android Version
SDK: 26 (8.0.0)

Caching

Caching

Fresco stores images in three different types of caches, organized hierarchically, with the cost of retrieving an image increasing the deeper you go.

1. Bitmap cache

The bitmap cache stores decoded images as Android `Bitmap` objects. These are ready for display or postprocessing.

On Android 4.x and lower, the bitmap cache's data lives in the *ashmem* heap, not in the Java heap. This means that images don't force extra runs of the garbage collector, slowing down your app.

Android 5.0 and newer has much improved memory management than earlier versions, so it is safer to leave the bitmap cache on the Java heap.

Your app should clear this cache when it is backgrounded.

2. Encoded memory cache

This cache stores images in their original compressed form. Images retrieved from this cache must be decoded before display.

If other transformations, such as resizing, rotation or transcoding were requested, that happens before decode.

3. Disk cache

(Yes, we know phones don't have disks, but it's too tedious to keep saying *local storage cache*…)

Like the encoded memory cache, this cache stores compressed image, which must be decoded and sometimes transformed before display.

Unlike the others, this cache is not cleared when your app exits, or even if the device is turned off.

When disk cache is about to be to the size limits defined by DiskCacheConfig Fresco uses LRU logic of eviction in disk cache (see DefaultEntryEvictionComparatorSupplier.java).

The user can, of course, always clear it from Android's Settings menu.

Checking to see if an item is in cache

You can use the methods in ImagePipeline to see if an item is in cache. The check for the memory cache is synchronous:

```
1 ImagePipeline imagePipeline = Fresco.getImagePipeline();
2 Uri uri;
3 boolean inMemoryCache = imagePipeline.isInBitmapMemoryCache(uri);
```

The check for the disk cache is asynchronous, since the disk check must be done on another thread. You can call it like this:

```
1  DataSource<Boolean> inDiskCacheSource = imagePipeline.isInDiskCache(uri);
2  DataSubscriber<Boolean> subscriber = new BaseDataSubscriber<Boolean>() {
3      @Override
4      protected void onNewResultImpl(DataSource<Boolean> dataSource) {
5        if (!dataSource.isFinished()) {
6          return;
7        }
8        boolean isInCache = dataSource.getResult();
9        // your code here
10     }
11   };
12 inDiskCacheSource.subscribe(subscriber, executor);
```

This assumes you are using the default cache key factory. If you have configured a custom one, you may need to use the methods that take an `ImageRequest` argument instead.

Evicting from cache

ImagePipeline also has methods to evict individual entries from cache:

```
1 ImagePipeline imagePipeline = Fresco.getImagePipeline();
2 Uri uri;
3 imagePipeline.evictFromMemoryCache(uri);
4 imagePipeline.evictFromDiskCache(uri);
5
6 // combines above two lines
7 imagePipeline.evictFromCache(uri);
```

As above, `evictFromDiskCache(Uri)` assumes you are using the default cache key factory. Users with a custom factory should use `evictFromDiskCache(ImageRequest)` instead.

Clearing the cache

```
1 ImagePipeline imagePipeline = Fresco.getImagePipeline();
2 imagePipeline.clearMemoryCaches();
3 imagePipeline.clearDiskCaches();
4
5 // combines above two lines
6 imagePipeline.clearCaches();
```

Using one disk cache or two?

Most apps need only a single disk cache. But in some circumstances you may want to keep smaller images in a separate cache, to prevent them from getting evicted too soon by larger images.

To do this, just call both `setMainDiskCacheConfig` and `setSmallImageDiskCacheConfig` methods when configuring the image pipeline.

What defines *small?* Your app does. When making an image request, you set its CacheChoice:

```
1 ImageRequest request = ImageRequestBuilder.newBuilderWithSource(uri)
2     .setCacheChoice(ImageRequest.CacheChoice.SMALL)
```

If you need only one cache, you can simply avoid calling `setSmallImageDiskCacheConfig`. The pipeline will default to using the same cache for both and `CacheChoice` will be ignored.

Trimming the caches

When configuring the image pipeline, you can set the maximum size of each of the caches. But there are times when you might want to go lower than that. For instance, your application might have caches for other kinds of data that might need more space and crowd out Fresco's. Or you might be checking to see if the device as a whole is running out of storage space.

Fresco's caches implement the DiskTrimmable or MemoryTrimmable interfaces. These are hooks into which your app can tell them to do emergency evictions.

Your application can then configure the pipeline with objects implementing the DiskTrimmableRegistry and MemoryTrimmableRegistry interfaces.

These objects must keep a list of trimmables. They must use app-specific logic to determine when memory or disk space must be preserved. They then notify the trimmable objects to carry out their trims.

Closeable References

Closeable References

This page is intended for advanced usage only.

Most apps should use Drawees and not worry about closing.

The Java language is garbage-collected and most developers are used to creating objects willy-nilly and taking it for granted they will eventually disappear from memory.

Until Android 5.0's improvements, this was not at all a good idea for Bitmaps. They take up a large share of the memory of a mobile device. Their existence in memory would make the garbage collector run more frequently, making image-heavy apps slow and janky.

Bitmaps were the one thing that makes Java developers miss C++ and its many smart pointer libraries, such as Boost.

Fresco's solution is found in the CloseableReference class. In order to use it correctly, you must follow the rules below:

1. The caller owns the reference.

Here, we create a reference, but since we're passing it to the caller, the caller takes the ownership:

```
CloseableReference<Val> foo() {
  Val val;
  // We are returning the reference from this method,
  // so whoever is calling this method is the owner
  // of the reference and is in charge of closing it.
  return CloseableReference.of(val);
}
```

2. The owner must close the reference before leaving scope.

Here we create a reference, but are not passing it to a caller. So we must close it:

```
 void gee() {
   // We are the caller of `foo` and so
   // we own the returned reference.
   CloseableReference<Val> ref = foo();
   try {
     // `haa` is a callee and not a caller, and so
     // it is NOT the owner of this reference, and
     // it must NOT close it.
     haa(ref);
   } finally {
     // We are not returning the reference to the
     // caller of this method, so we are still the owner,
     // and must close it before leaving the scope.
     ref.close();
   }
}
```

The `finally` block is almost always the best way to do this.

3. Never close the value.

`CloseableReference` wraps a shared resource which gets released when there are no more active references pointing to it. Tracking of active references is done automatically by an internal reference counter. When the reference count drops to 0, `CloseableReference` will release the underlying resource. The very purpose of `CloseableReference` is to manage the underlying resource so that you don't have to. That said, you are responsible for closing the `CloseableReference` if you own it, but **not** the value it points to! If you explicitly close the underlying value, you will erroneously invalidate all the other active references pointing to that same resource.

```
1    CloseableReference<Val> ref = foo();
2
3    Val val = ref.get();
4    // do something with val
5    // ...
6
7    // Do NOT close the value!
8    //// val.close();
9
10   // DO close the reference instead.
11   ref.close();
```

4. Something other than the owner should *not* close the reference.

Here, we are receiving the reference via argument. The caller is still the owner, so we are not supposed to close it.

```
1 void haa(CloseableReference<?> ref) {
2   // We are callee, and not a caller, and so
3   // we must NOT close the reference.
4   // We are guaranteed that the reference won't
5   // become invalid for the duration of this call.
6   Log.println("Haa: " + ref.get());
7 }
```

If we called `.close()` here by mistake, then if the caller tried to call `.get()`, an `IllegalStateException` would be thrown.

5. Callee should always clone the reference before assigning.

If we need to hold onto the reference, we need to clone it.

If using it in a class:

```
1  class MyClass {
2    CloseableReference<Val> myValRef;
3
4    void mmm(CloseableReference<Val> ref) {
5      // Some caller called this method. Caller owns the original
6      // reference and if we want to have our own copy, we must clone it.
7      myValRef = ref.clone();
8    };
9    // caller can now safely close its copy as we made our own clone.
10
11   void close() {
12     // We are in charge of closing our copy, of course.
13     CloseableReference.closeSafely(myValRef);
14   }
15 }
16 // Now the caller of MyClass must close it!
```

If using it in an inner class:

```
1  void haa(CloseableReference<?> ref) {
2    // Here we make our own copy of the original reference,
3    // so that we can guarantee its validity when the executor
4    // executes our runnable in the future.
5    final CloseableReference<?> refClone = ref.clone();
6    executor.submit(new Runnable() {
7      public void run() {
8        try {
9          Log.println("Haa Async: " + refClone.get());
10       } finally {
11         // We need to close our copy once we are done with it.
12         refClone.close();
13       }
14     }
15   });
16   // caller can now safely close its copy as we made our own clone.
17 }
```

Configuring the Image Pipeline

Configuring the Image Pipeline

Most apps can initialize Fresco completely by the simple command:

```
1 Fresco.initialize(context);
```

For those apps that need more advanced customization, we offer it using the ImagePipelineConfig class.

Here is a maximal example. Rare is the app that actually needs all of these settings, but here they are for reference.

```
1  ImagePipelineConfig config = ImagePipelineConfig.newBuilder(context)
2      .setBitmapMemoryCacheParamsSupplier(bitmapCacheParamsSupplier)
3      .setCacheKeyFactory(cacheKeyFactory)
4      .setDownsampleEnabled(true)
5      .setWebpSupportEnabled(true)
6      .setEncodedMemoryCacheParamsSupplier(encodedCacheParamsSupplier)
7      .setExecutorSupplier(executorSupplier)
8      .setImageCacheStatsTracker(imageCacheStatsTracker)
9      .setMainDiskCacheConfig(mainDiskCacheConfig)
10     .setMemoryTrimmableRegistry(memoryTrimmableRegistry)
11     .setNetworkFetchProducer(networkFetchProducer)
12     .setPoolFactory(poolFactory)
13     .setProgressiveJpegConfig(progressiveJpegConfig)
14     .setRequestListeners(requestListeners)
15     .setSmallImageDiskCacheConfig(smallImageDiskCacheConfig)
16     .build();
17 Fresco.initialize(context, config);
```

Be sure to pass your `ImagePipelineConfig` object to `Fresco.initialize!` Otherwise, Fresco will use a default configuration instead of the one you built.

Understanding Suppliers

Several of the configuration builder's methods take arguments of a Supplier of an instance rather than an instance itself. This is a little more complex to create, but allows you to change behaviors while your app is running. Memory caches, for one, check their Supplier every five minutes.

If you don't need to dynamically change the params, use a Supplier that returns the same object each time:

```
1 Supplier<X> xSupplier = new Supplier<X>() {
2   private X mX = new X(xparam1, xparam2...);
3   public X get() {
4     return mX;
5   }
6 );
7 // when creating image pipeline
8 .setXSupplier(xSupplier);
```

Thread pools

By default, the image pipeline uses three thread pools:

1. Three threads for network downloads
2. Two threads for all disk operations - local file reads, and the disk cache
3. Two threads for all CPU-bound operations - decodes, transforms, and background operations, such as postprocessing.

You can customize networking behavior by setting your own network layer.

To change the behavior for all other operations, pass in an instance of ExecutorSupplier.

Using a MemoryTrimmableRegistry

If your application listens to system memory events, it can pass them over to Fresco to trim memory caches.

The easiest way for most apps to listen to events is to override Activity.onTrimMemory. You can also use any subclass of ComponentCallbacks2.

You should have an implementation of MemoryTrimmableRegistry. This object should keep a collection of MemoryTrimmable objects - Fresco's caches will be among them. When getting a system memory event, you call the appropriate `MemoryTrimmable` method on each of the trimmables.

Configuring the memory caches

The bitmap cache and the encoded memory cache are configured by a Supplier of a MemoryCacheParams object.

Configuring the disk cache

You use the builder pattern to create a DiskCacheConfig object:

```
1 DiskCacheConfig diskCacheConfig = DiskCacheConfig.newBuilder()
2    .set....
3    .set....
4    .build()
5
6 // when building ImagePipelineConfig
7 .setMainDiskCacheConfig(diskCacheConfig)
```

Keeping cache stats

If you want to keep track of metrics like the cache hit rate, you can implement the ImageCacheStatsTracker class. This provides callbacks for every cache event that you can use to keep your own statistics.

Customizing Image Formats

Customizing Image Formats

In general, two parts are involved until an image can be displayed on screen:

1. decoding the image
2. rendering the decoded image

Fresco allows you to customize both of these parts. For example, it's possible to add a custom image decoder for an existing image format or for a new image format that uses Fresco's built-in rendering architecture to render bitmaps. Or, it's possible to let the built-in decoder handle decoding and then create a custom Drawable used to render the image on screen. And, of course, you can also do both. These customizations can be either registered globally when Fresco is initialized or locally for selected images only.

The (much simplified) decoding and rendering process looks like this:

1. The encoded image is downloaded from the network or loaded from the disk cache.
2. The `ImageFormat` of the `EncodedImage` is determined using a class called `ImageFormatChecker`, which has a list of `ImageFormat.FormatChecker` objects, one for each recognized image format.
3. The `EncodedImage` is decoded using a suitable `ImageDecoder` for the given format and returns an object that extends `CloseableImage`, which represents the decoded image.
4. From a list of `DrawableFactory` objects, the first one that is able to handle the `CloseableImage` is used to create a `Drawable`.
5. The drawable is rendered on screen.

It is possible to add custom image formats by adding an `ImageFormat.FormatChecker` for step 2. You can supply custom `ImageDecoder`s to add decoding support for new image formats or override built-in decoding. Finally, you can supply a custom `DrawableFactory` to use a custom `Drawable` for rendering the image.

All default image formats can be found in `DefaultImageFormats` and `DefaultImageFormatChecker`, the default drawable factory is in `PipelineDraweeController` and several samples for customizing them can be found in the Showcase sample app.

Custom decoders

Let's start with an example. In order to create a custom decoder, simply implement the `ImageDecoder` interface:

```
1  public class CustomDecoder implements ImageDecoder {
2
3    @Override
4    public CloseableImage decode(
5        EncodedImage encodedImage,
6        int length,
7        QualityInfo qualityInfo,
8        ImageDecodeOptions options) {
9      // Decode the given encodedImage and return a
10     // corresponding (decoded) CloseableImage.
11     CloseableImage closeableImage = ...;
12     return closeableImage;
13   }
14 }
```

The given encoded image can be used to return a class that extends `CloseableImage`, which represents the decoded image and which will then be automatically cached for you. You can either return one of the existing `CloseableImage` types, like `CloseableStaticBitmap` for bitmaps, or define your own `CloseableImage` class.

Custom decoders can be set globally or locally on a per-image basis. For local overrides, you can set the custom decoder as follows:

```
ImageDecoder customDecoder = ...;
Uri uri = ...;
draweeView.setController(
  Fresco.newDraweeControllerBuilder()
        .setImageRequest(
          ImageRequestBuilder.newBuilderWithSource(uri)
              .setImageDecodeOptions(
                  ImageDecodeOptions.newBuilder()
                      .setCustomImageDecoder(customDecoder)
                      .build())
              .build())
        .build());
```

NOTE: If you're supplying a custom decoder, it will be used for all images. The default decoder will be completely bypassed.

Custom image formats

You simply create a new `ImageFormat` object and hold on to it in your code:

```
private static final ImageFormat CUSTOM_FORMAT = new ImageFormat("format name", "format file extension");
```

All supported default image formats can be found in `DefaultImageFormats`.

Then, we need to create a custom `ImageFormat.FormatChecker` that is used to detect your new image format. The format checker has 2 methods, one to determine the number of header bytes required to make the decision (keep this number as small as possible since this operation is performed for all images) and the actual `determineFormat` method, which should return **the same `ImageFormat` instance**, `CUSTOM_FORMAT` in this example - or `null` if the image is of a different format. A simple format checker could look like this:

```
public static class ColorFormatChecker implements ImageFormat.FormatChecker {

  private static final byte[] HEADER = ImageFormatCheckerUtils.asciiBytes("my_header");

  @Override
  public int getHeaderSize() {
    return HEADER.length;
  }

  @Nullable
  @Override
  public ImageFormat determineFormat(byte[] headerBytes, int headerSize) {
    if (headerSize < getHeaderSize()) {
      return null;
    }
    if (ImageFormatCheckerUtils.startsWithPattern(headerBytes, HEADER)) {
      return CUSTOM_FORMAT;
    }
    return null;
  }
}
```

The third component required for custom image format is a custom decoder as explained above that can create the actual decoded image.

You have to register your custom image format with Fresco by supplying a `ImageDecoderConfig` to Fresco when it is initialized. Similarly, you can override the default decoding behavior by using a built-in image format:

```
ImageFormat myFormat = ...;
ImageFormat.FormatChecker myFormatChecker = ...;
ImageDecoder myDecoder = ...;
ImageDecoderConfig imageDecoderConfig = new ImageDecoderConfig.Builder()
  .addDecodingCapability(
    myFormat,
    myFormatChecker,
    myDecoder)
  .build();

ImagePipelineConfig config = ImagePipelineConfig.newBuilder()
  .setImageDecoderConfig(imageDecoderConfig)
  .build();

Fresco.initialize(context, config);
```

Custom drawables

If a `DraweeController` is used to load the image (e.g. if you're using a `DraweeView`), a corresponding `DrawableFactory` is used to create a drawable to render the decoded image based on the `CloseableImage`. If you're manually using the image pipeline, you have to handle the `CloseableImage` itself.

If you use one of the built-in types, like `CloseableStaticBitmap`, the `PipelineDraweeController` already knows how to handle the format and will create a `BitmapDrawable` for you. If you want to override that behavior or add support for custom `CloseableImage`s, you have to implement a drawable factory:

```
public static class CustomDrawableFactory implements DrawableFactory {

  @Override
  public boolean supportsImageType(CloseableImage image) {
    // You can either override a built-in format, like `CloseableStaticBitmap`
    // or your own implementations.
    return image instanceof CustomCloseableImage;
  }

  @Nullable
  @Override
  public Drawable createDrawable(CloseableImage image) {
    // Create and return your custom drawable for the given CloseableImage.
    // It is guaranteed that the `CloseableImage` is an instance of the
    // declared classes in `supportsImageType` above.
    CustomCloseableImage myCloseableImage = (CustomCloseableImage) image;
    Drawable myDrawable = ...; //e.g. new CustomDrawable(myCloseableImage)
    return myDrawable;
  }
}
```

In order to use your drawable factory, you can either use a global or local override.

Global custom drawable override

You have to register all global drawable factories when Fresco is initialized:

```
DrawableFactory myDrawableFactory = ...;

DraweeConfig draweeConfig = DraweeConfig.newBuilder()
  .addCustomDrawableFactory(myDrawableFactory)
  .build();

Fresco.initialize(this, imagePipelineConfig, draweeConfig);
```

Local custom drawable override

For local overrides, the `PipelineDraweeControllerBuilder` offers methods to set custom drawable factories:

```
DrawableFactory myDrawableFactory = ...;
Uri uri = ...;

simpleDraweeView.setController(Fresco.newDraweeControllerBuilder()
  .setUri(uri)
  .setCustomDrawableFactory(factory)
  .build());
```

DataSources and DataSubscribers

DataSources and DataSubscribers

A DataSource is, like a Java Future, the result of an asynchronous computation. The different is that, unlike a Future, a DataSource can return you a whole series of results from a single command, not just one.

After submitting an image request, the image pipeline returns a data source. To get a result out if it, you need to use a DataSubscriber.

Executors

When subscribing to a data source, an executor must be provided. The purpose of executors is to execute runnables (in our case the subscriber callback methods) on a specific thread and with specific policy. Fresco provides several executors and one should carefully choose which one to be used:

- If you need to do any UI stuff from your callback (accessing views, drawables, etc.), you must use `UiThreadImmediateExecutorService.getInstance()`. Android view system is not thread safe and is only to be accessed from the main thread (the UI thread).
- If the callback is lightweight, and does not do any UI related stuff, you can simply use `CallerThreadExecutor.getInstance()`. This executor executes runnables on the caller's thread. Depending on what is the calling thread, callback may be executed either on the UI or a background thread. There are no guarantees which thread it is going to be and because of that this executor should be used with great caution. And again, only for lightweight non-UI related stuff.
- If you need to do some expensive non-UI related work (database access, disk read/write, or any other slow operation), this should NOT be done either with `CallerThreadExecutor` nor with the `UiThreadExecutorService`, but with one of the background thread executors. See DefaultExecutorSupplier.forBackgroundTasks for an example implementation.

Getting result from a data source

This is a generic example of how to get a result from a data source of `CloseableReference<T>` for arbitrary type `T`. The result is valid only in the scope of the `onNewResultImpl` callback. As soon as the callback gets executed, the result is no longer valid. See the next example if the result needs to be kept around.

```
1      DataSource<CloseableReference<T>> dataSource = ...;
2
3      DataSubscriber<CloseableReference<T>> dataSubscriber =
4          new BaseDataSubscriber<CloseableReference<T>>() {
5            @Override
6            protected void onNewResultImpl(
7                DataSource<CloseableReference<T>> dataSource) {
8              if (!dataSource.isFinished()) {
9                // if we are not interested in the intermediate images,
10               // we can just return here.
11               return;
12             }
13             CloseableReference<T> ref = dataSource.getResult();
14             if (ref != null) {
15               try {
16                 // do something with the result
17                 T result = ref.get();
18                 ...
19               } finally {
20                 CloseableReference.closeSafely(ref);
21               }
22             }
23           }
24
25           @Override
26           protected void onFailureImpl(DataSource<CloseableReference<T>> dataSource) {
27             Throwable t = dataSource.getFailureCause();
28             // handle failure
29           }
30         };
31
32     dataSource.subscribe(dataSubscriber, executor);
```

Keeping result from a data source

The above example closes the reference as soon as the callback gets executed. If the result needs to be kept around, you must keep the corresponding `CloseableReference` for as long as the result is needed. This can be done as follows:

```
1      DataSource<CloseableReference<T>> dataSource = ...;
2
3      DataSubscriber<CloseableReference<T>> dataSubscriber =
4          new BaseDataSubscriber<CloseableReference<T>>() {
5            @Override
6            protected void onNewResultImpl(
7                DataSource<CloseableReference<T>> dataSource) {
8              if (!dataSource.isFinished()) {
9                // if we are not interested in the intermediate images,
10               // we can just return here.
11               return;
12             }
13             // keep the closeable reference
14             mRef = dataSource.getResult();
15             // do something with the result
16             T result = mRef.get();
17             ...
18           }
19
20           @Override
21           protected void onFailureImpl(DataSource<CloseableReference<T>> dataSource) {
22             Throwable t = dataSource.getFailureCause();
23             // handle failure
24           }
25         };
26
27     dataSource.subscribe(dataSubscriber, executor);
```

IMPORTANT: once you don't need the result anymore, you **must close the reference**. Not doing so may cause memory leaks. See closeable references for more details.

```
1     CloseableReference.closeSafely(mRef);
2     mRef = null;
```

However, if you are using `BaseDataSubscriber` you do not have to manually close the `dataSource` (closing `mRef` is enough). `BaseDataSubscriber` automatically closes the `dataSource` for you right after `onNewResultImpl` is called. If you are not using `BaseDataSubscriber` (e.g. if you're calling `dataSource.getResult()`), make sure to close the `dataSource` as well.

To get encoded image…

```
1     DataSource<CloseableReference<PooledByteBuffer>> dataSource =
2         mImagePipeline.fetchEncodedImage(imageRequest, CALLER_CONTEXT);
```

Image pipeline uses `PooledByteBuffer` for encoded images. This is our `T` in the above examples. Here is an example of creating an `InputStream` out of `PooledByteBuffer` so that we can read the image bytes:

```
1        InputStream is = new PooledByteBufferInputStream(result);
2        try {
3          // Example: get the image format
4          ImageFormat imageFormat = ImageFormatChecker.getImageFormat(is);
5          // Example: write input stream to a file
6          Files.copy(is, path);
7        } catch (...) {
8          ...
9        } finally {
10         Closeables.closeQuietly(is);
11       }
```

To get decoded image…

```
DataSource<CloseableReference<CloseableImage>>
    dataSource = imagePipeline.fetchDecodedImage(imageRequest, callerContext);
```

Image pipeline uses `CloseableImage` for decoded images. This is our `T` in the above examples. Here is an example of getting a `Bitmap` out of `CloseableImage`:

```
CloseableImage image = ref.get();
if (image instanceof CloseableBitmap) {
  // do something with the bitmap
  Bitmap bitmap = (CloseableBitmap image).getUnderlyingBitmap();
  ...
}
```

I just want a bitmap…

If your request to the pipeline is for a single Bitmap, you can take advantage of our easier-to-use BaseBitmapDataSubscriber:

```
dataSource.subscribe(new BaseBitmapDataSubscriber() {
    @Override
    public void onNewResultImpl(@Nullable Bitmap bitmap) {
      // You can use the bitmap here, but in limited ways.
      // No need to do any cleanup.
    }

    @Override
    public void onFailureImpl(DataSource dataSource) {
      // No cleanup required here.
    }
  },
  executor);
```

A snap to use, right? There are caveats.

This subscriber doesn't work for animated images as those can not be represented as a single bitmap.

You can **not** assign the bitmap to any variable not in the scope of the `onNewResultImpl` method. The reason is, as already explained in the above examples that, after the subscriber has finished executing, the image pipeline will recycle the bitmap and free its memory. If you try to draw the bitmap after that, your app will crash with an `IllegalStateException`.

You can still safely pass the Bitmap to an Android notification or remote view. If Android needs your Bitmap in order to pass it to a system process, it makes a copy of the Bitmap data in ashmem - the same heap used by Fresco. So Fresco's automatic cleanup will work without issue.

If those requirements prevent you from using `BaseBitmapDataSubscriber`, you can go with a more generic approach as explained above.

Image Requests

Image Requests

If you need an `ImageRequest` that consists only of a URI, you can use the helper method `ImageRequest.fromURI`. Loading multiple-images is a common case of this.

If you need to tell the image pipeline anything more than a simple URI, you need to use `ImageRequestBuilder`:

```
1  Uri uri;
2
3  ImageDecodeOptions decodeOptions = ImageDecodeOptions.newBuilder()
4      .setBackgroundColor(Color.GREEN)
5      .build();
6
7  ImageRequest request = ImageRequestBuilder
8      .newBuilderWithSource(uri)
9      .setImageDecodeOptions(decodeOptions)
10     .setAutoRotateEnabled(true)
11     .setLocalThumbnailPreviewsEnabled(true)
12     .setLowestPermittedRequestLevel(RequestLevel.FULL_FETCH)
13     .setProgressiveRenderingEnabled(false)
14     .setResizeOptions(new ResizeOptions(width, height))
15     .build();
```

Fields in ImageRequest

- `uri` - the only mandatory field. See Supported URIs
- `autoRotateEnabled` - whether to enable auto-rotation.
- `progressiveEnabled` - whether to enable progressive loading.
- `postprocessor` - component to postprocess the decoded image.
- `resizeOptions` - desired width and height. Use with caution. See Resizing.

Lowest Permitted Request Level

The image pipeline follows a definite sequence in where it looks for the image.

1. Check the bitmap cache. This is nearly instant. If found, return.
2. Check the encoded memory cache. If found, decode the image and return.
3. Check the “disk” (local storage) cache. If found, load from disk, decode, and return.
4. Go to the original file on network or local file. Download, resize and/or rotate if requested, decode, and return. For network images in particular, this will be the slowest by a long shot.

The `setLowestPermittedRequestLevel` field lets you control how far down this list the pipeline will go. Possible values are:

- `BITMAP_MEMORY_CACHE`
- `ENCODED_MEMORY_CACHE`
- `DISK_CACHE`
- `FULL_FETCH`

This is useful in situations where you need an instant, or at least relatively fast, image or none at all.

Images in Notifications

Images in Notifications

If you need to display an image in a notification, you can use the `BaseBitmapDataSubscriber` for requesting a bitmap from the `ImagePipeline`. This is safe to be passed to a notification as the system will parcel it after the `NotificationManager#notify` method. This page explains a full sample on how to do this.

Step by step

First create an `ImageRequest` with the URI:

```
1 ImageRequest imageRequest = ImageRequest.fromUri("http://example.org/user/42/profile.jpg"));
```

Then create a `DataSource` and request the decoded image from the `ImagePipeline`:

```
1 ImagePipeline imagePipeline = Fresco.getImagePipeline();
2 DataSource<CloseableReference<CloseableImage>> dataSource = imagePipeline.fetchDecodedImage(imageRequest, null);
```

As a `DataSource` is similar to a `Future`, we need to add a `DataSubscriber` to handle the result. The `BaseBitmapDataSubscriber` abstracts some of the complexity away when dealing with `Bitmap`:

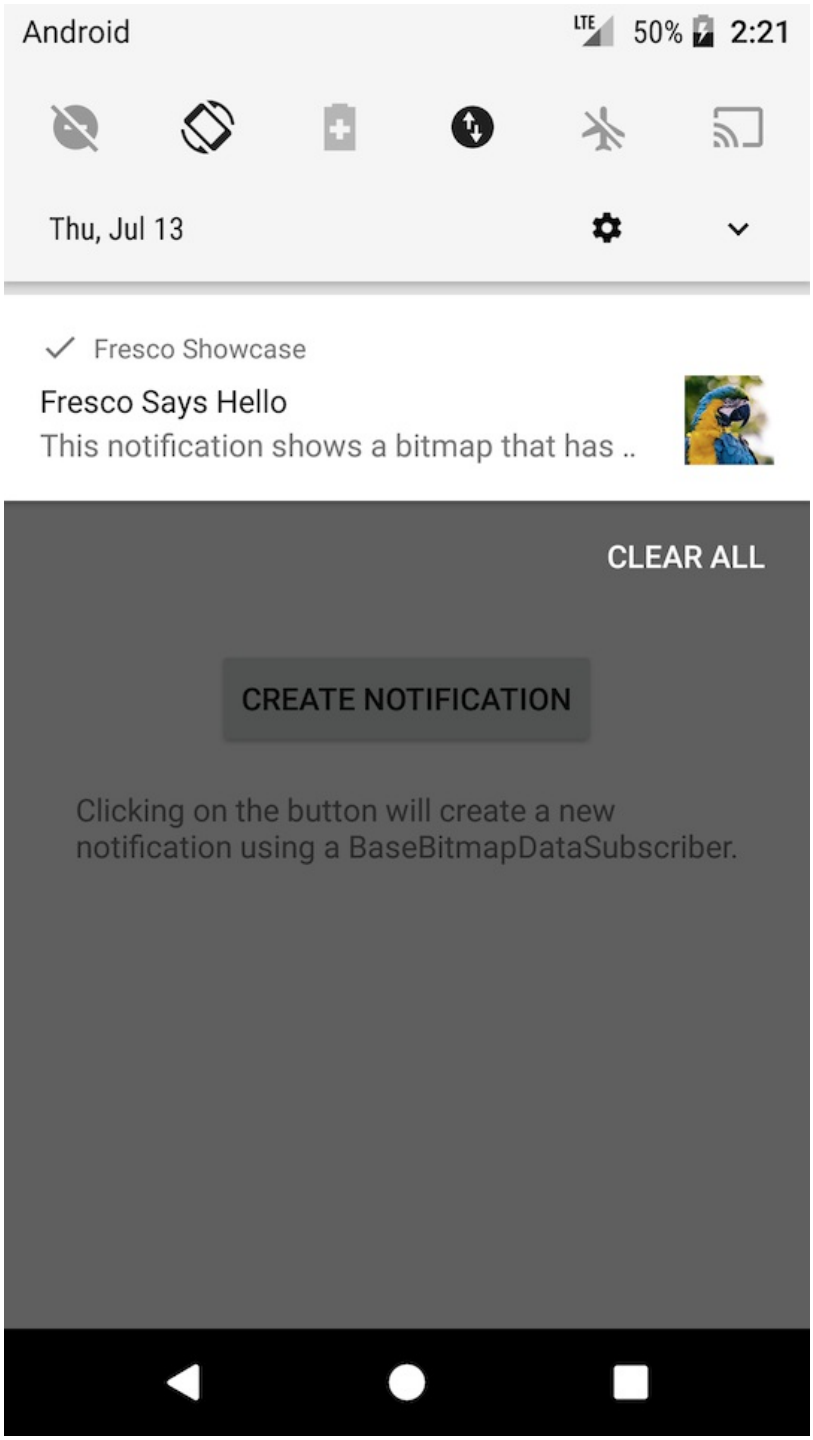
```
1  dataSource.subscribe(
2      new BaseBitmapDataSubscriber() {
3
4        @Override
5        protected void onNewResultImpl(Bitmap bitmap) {
6          displayNotification(bitmap);
7        }
8
9        @Override
10       protected void onFailureImpl(DataSource<CloseableReference<CloseableImage>> dataSource) {
11         // In general, failing to fetch the image should not keep us from displaying the
12         // notification. We proceed without the bitmap.
13         displayNotification(null);
14       }
15     },
16     UiThreadImmediateExecutorService.getInstance());
17 }
```

The `displayNotification(Bitmap)` method then is similar to the 'normal' way to do this on Android:

```
1  private void displayNotification(@Nullable Bitmap bitmap) {
2    final NotificationCompat.Builder notificationBuilder =
3        new NotificationCompat.Builder(getContext())
4            .setSmallIcon(R.drawable.ic_done)
5            .setLargeIcon(bitmap)
6            .setContentTitle("Fresco Says Hello")
7            .setContentText("Notification Text ...");
8
9    final NotificationManager notificationManager =
10       (NotificationManager) getContext().getSystemService(Context.NOTIFICATION_SERVICE);
11
12   notificationManager.notify(NOTIFICATION_ID, notificationBuilder.build());
13 }
```

Full Sample

For the full sample see the `ImagePipelineNotificationFragment` in the showcase app: ImagePipelineNotificationFragment.java


Android
LTE 50% 2:21
Thu, Jul 13
Fresco Showcase
Fresco Says Hello
This notification shows a bitmap that has ..
CLEAR ALL
CREATE NOTIFICATION
Clicking on the button will create a new notification using a BaseBitmapDataSubscriber.

Listening to Events

Listening to Events

Motivation

The image pipeline and the view controller in Fresco have built-in instrumentation interfaces. One can employ this to track both performance and to react to events.

Fresco comes with two main instrumentation interfaces:

- The `RequestListener` is globally registered in the `ImagePipelineConfig` and logs all requests that are handled by the producer-consumer chain
- The `ControllerListener` is added to an individual `DraweeView` and is convenient for reacting on events such as "this image is fully loaded"

ControllerListener

While the `RequestListener` is a global listener, the `ControllerListener` is local to a certain `DraweeView`. It is a good way to react to changes to the displayed view such as "image failed to load" or "image is fully loaded". Again, it's best to extend `BaseControllerListener` for this.

A simple listener might look like the following:

```
1  public class MyControllerListener extends new BaseControllerListener<ImageInfo>() {
2
3    @Override
4    public void onFinalImageSet(String id, ImageInfo imageInfo, Animatable animatable) {
5      Log.i("DraweeUpdate", "Image is fully loaded!");
6    }
7
8    @Override
9    public void onIntermediateImageSet(String id, ImageInfo imageInfo, Animatable animatable) {
10     Log.i("DraweeUpdate", "Image is partly loaded! (maybe it's a progressive JPEG?)");
11     if (imageInfo != null) {
12       int quality = imageInfo.getQualityInfo().getQuality();
13       Log.i("DraweeUpdate", "Image quality (number scans) is: " + quality);
14     }
15   }
16
17   @Override
18   public void onFailure(String id, Throwable throwable) {
19     Log.i("DraweeUpdate", "Image failed to load: " + throwable.getMessage());
20   }
21 }
```

You add it to your `DraweeController` in the following way:

```
1 DraweeController controller = Fresco.newDraweeControllerBuilder()
2     .setImageRequest(request)
3     .setControllerListener(new MyControllerListener())
4     .build();
5 mSimpleDraweeView.setController(controller);
```

RequestListener

The `RequestListener` comes with a large interface of callback methods. Most importantly, you will notice that they all provide the unique `requestId` which allows to track a request across multiple stages.

Due to the large number of callbacks, it is advisable to extend from `BaseRequestListener` instead and only implement the methods you are interested in. You register your listener in the Application class as follows:

```
1 final Set<RequestListener> listeners = new HashSet<>();
2 listeners.add(new MyRequestLoggingListener());
3
4 ImagePipelineConfig imagePipelineConfig = ImagePipelineConfig.newBuilder(this)
5   .setRequestListeners(listeners)
6   .build();
7
8 Fresco.initialize(this, imagePipelineConfig);
```

We will walk through the generated logging of one image request from the showcase app and discuss the individual meanings. You can observe these yourself in `adb logcat` when running the showcase app:

```
1 RequestLoggingListener: time 2095589: onRequestSubmit: {requestId: 5, callerContext: null, isPrefetch: false}
```

`onRequestSubmit(...)` is called when an `ImageRequest` enters the image pipeline. Here you can make use of the caller context object to identify which feature of the app is sending the request.

```
1 RequestLoggingListener: time 2095590: onProducerStart: {requestId: 5, producer: BitmapMemoryCacheGetProducer}
2 RequestLoggingListener: time 2095591: onProducerFinishWithSuccess: {requestId: 5, producer: BitmapMemoryCacheGetProducer, elapsedTime: 1 ms, extraMap: {cached_value_foun
```

The `onProducerStart(...)` and `onProducerFinishWithSuccess(...)` (or `onProducerFinishWithFailure(...)`) are called for all producers along the pipeline. The one above is a check of the Bitmap cache.

```
1  RequestLoggingListener: time 2095592: onProducerStart: {requestId: 5, producer: BackgroundThreadHandoffProducer}
2  RequestLoggingListener: time 2095593: onProducerFinishWithSuccess: {requestId: 5, producer: BackgroundThreadHandoffProducer, elapsedTime: 1 ms, extraMap: null}
3  RequestLoggingListener: time 2095594: onProducerStart: {requestId: 5, producer: BitmapMemoryCacheProducer}
4  RequestLoggingListener: time 2095594: onProducerFinishWithSuccess: {requestId: 5, producer: BitmapMemoryCacheProducer, elapsedTime: 0 ms, extraMap: {cached_value_found=
5  RequestLoggingListener: time 2095595: onProducerStart: {requestId: 5, producer: EncodedMemoryCacheProducer}
6  RequestLoggingListener: time 2095596: onProducerFinishWithSuccess: {requestId: 5, producer: EncodedMemoryCacheProducer, elapsedTime: 1 ms, extraMap: {cached_value_found
7  RequestLoggingListener: time 2095596: onProducerStart: {requestId: 5, producer: DiskCacheProducer}
8  RequestLoggingListener: time 2095598: onProducerFinishWithSuccess: {requestId: 5, producer: DiskCacheProducer, elapsedTime: 2 ms, extraMap: {cached_value_found=false}}
9  RequestLoggingListener: time 2095598: onProducerStart: {requestId: 5, producer: PartialDiskCacheProducer}
10 RequestLoggingListener: time 2095602: onProducerFinishWithSuccess: {requestId: 5, producer: PartialDiskCacheProducer, elapsedTime: 4 ms, extraMap: {cached_value_found=f
```

We see more of these when the request is handed over to the background (`BackgroundThreadHandoffProducer`) and performs look-ups in the caches.

```
1 RequestLoggingListener: time 2095602: onProducerStart: {requestId: 5, producer: NetworkFetchProducer}
2 RequestLoggingListener: time 2095745: onProducerEvent: {requestId: 5, stage: NetworkFetchProducer, eventName: intermediate_result; elapsedTime: 143 ms}
3 RequestLoggingListener: time 2095764: onProducerFinishWithSuccess: {requestId: 5, producer: NetworkFetchProducer, elapsedTime: 162 ms, extraMap: {queue_time=140, total_t
4 RequestLoggingListener: time 2095764: onUltimateProducerReached: {requestId: 5, producer: NetworkFetchProducer, elapsedTime: -1 ms, success: true}
```

For this particular request, the `NetworkFetchProducer` is the "ultimate producer". This means, it is the one that provides the definite input source for fulfilling the request. If the image is cached, the `DiskCacheProducer` would be the "ultimate" producer.

```
1 RequestLoggingListener: time 2095766: onProducerStart: {requestId: 5, producer: DecodeProducer}
2 RequestLoggingListener: time 2095786: onProducerFinishWithSuccess: {requestId: 5, producer: DecodeProducer, elapsedTime: 20 ms, extraMap: {imageFormat=JPEG, ,hasGoodQual
3 RequestLoggingListener: time 2095788: onRequestSuccess: {requestId: 5, elapsedTime: 198 ms}
```

On the way up, the `DecodeProducer` also succeeds and finally the `onRequestSuccess(...)` method is called.

You will notice that most of these methods are given optional information as a `Map<String, String> extraMap`. The string constants to look-up the elements are usually public constants in the corresponding producer classes.

Prefetching Images

Prefetching Images

Prefetching images in advance of showing them can sometimes lead to shorter wait times for users. Remember, however, that there are trade-offs. Prefetching takes up your users' data, and uses up its share of CPU and memory. As a rule, prefetching is not recommended for most apps.

Nonetheless, the image pipeline allows you to prefetch to either disk or bitmap cache. Both will use more data for network URIs, but the disk cache will not do a decode and will therefore use less CPU.

Note: Beware that if your network fetcher doesn't support priorities prefetch requests may slow down images which are immediately required on screen. Neither `OkHttpNetworkFetcher` nor `HttpUrlConnectionNetworkFetcher` currently support priorities.

Prefetch to disk:

```
1 imagePipeline.prefetchToDiskCache(imageRequest, callerContext);
```

Prefetch to bitmap cache:

```
1 imagePipeline.prefetchToBitmapCache(imageRequest, callerContext);
```

Cancelling prefetch:

```
1 // keep the reference to the returned data source.
2 DataSource<Void> prefetchDataSource = imagePipeline.prefetchTo...;
3
4 // later on, if/when you need to cancel the prefetch:
5 prefetchDataSource.close();
```

Closing a prefetch data source after the prefetch has already completed is a no-op and completely safe to do.

Example

See our showcase app for a practical example of how to use prefetching.

Modifying the Image (Post-processing)

Modifying the Image (Post-processing)

Motivation

Post-processors allow custom modifications of the fetched image. In most cases image processing should already be done by the server before the image is sent down to the client, as the mobile device's resources are usually more limited. However, there are many instances where client side processing is a valid option. For instance, if the images are being served by a third party which you do not control or if the images are local (on the device).

Background

In Fresco's pipeline, post-processors are applied at the very end when the image already has been decoded as a bitmap and the original version is stored in the in-memory Bitmap cache. While the post-processor can directly work on the provided Bitmap, it can also create a new Bitmap with a different dimension.

Ideally, the implemented post-processor should provide a cache key for given parameters. By doing this, the newly generated bitmap is also cached in the in-memory Bitmap cache and don't need to be re-created.

All post-processors are executed using background executors. However, naive iteration or complex computations can still take a long time and should be avoided. If you aim for computations that are non-linear in the number of pixels, there is a section which contains tips for you how you can use native code to speed your post-processor up.

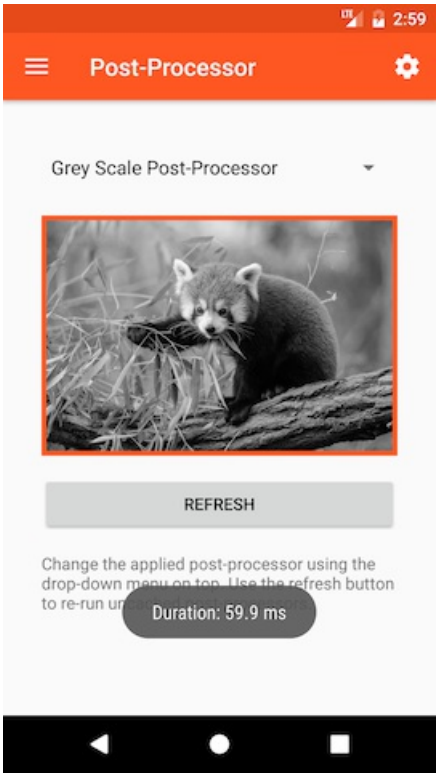
Example: Creating a Grey-Scale Filter

Let's start with something simple: a post-processor that converts the bitmap into a grey-scale version. For this we need to iterate over the bitmap's pixels and replace their color value.

The image is copied before it enters the post-processor. The original image in cache is *not* affected by any changes you make in your post-processor. On Android 4.x and lower, the copy is stored outside the Java heap, just as the original image was.

The `BasePostprocessor` expects our sub-class to override one of its `BasePostprocessor#process` method. The simplest one performs in-place modifications of the provided bitmap. Here, the image is copied before it enters the post-processor. Thus, the original of the image in cache is *not* affected by any changes you make in the post-processor. We will later discuss how we can also modify the configuration and size of the outputted bitmap.

```
1  public class FastGreyScalePostprocessor extends BasePostprocessor {
2
3    @Override
4    public void process(Bitmap bitmap) {
5      final int w = bitmap.getWidth();
6      final int h = bitmap.getHeight();
7      final int[] pixels = new int[w * h];
8
9      bitmap.getPixels(pixels, 0, w, 0, 0, w, h);
10
11     for (int x = 0; x < w; x++) {
12       for (int y = 0; y < h; y++) {
13         final int offset = y * w + x;
14         pixels[offset] = getGreyColor(pixels[offset]);
15       }
16     }
17
18     // this is much faster then calling #getPixel and #setPixel as it crosses
19     // the JNI barrier only once
20     bitmap.setPixels(pixels, 0, w, 0, 0, w, h);
21   }
22
23   static int getGreyColor(int color) {
24     final int alpha = color & 0xFF000000;
25     final int r = (color >> 16) & 0xFF;
26     final int g = (color >> 8) & 0xFF;
27     final int b = color & 0xFF;
28
29     // see: https://en.wikipedia.org/wiki/Relative_luminance
30     final int luminance = (int) (0.2126 * r + 0.7152 * g + 0.0722 * b);
31
32     return alpha | luminance << 16 | luminance << 8 | luminance;
33   }
34 }
```



Caching Post-Processor Results

As we've seen that post-processing computations can be rather resource intensive, we want to cache the results. Cached output bitmaps are stored in the same cache as the decoded input bitmaps.

In order to use this feature, the post-processor must override the `PostProcessor#getPostProcessorCacheKey` method. It should return a cache key that is dependent on all important input values that effect the performed modifications.

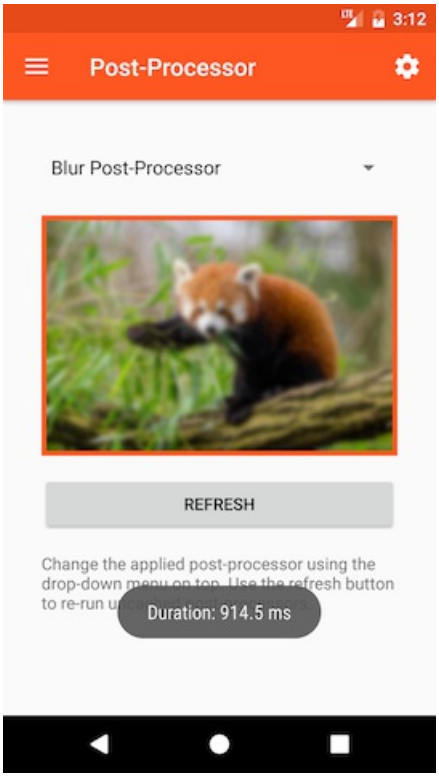
For this example we extend an existing `WatermarkPostprocessor` that draws a watermark text multiple times on the image:

```
1  public class CachedWatermarkPostprocessor extends WatermarkPostprocessor {
2
3    @Override
4    public CacheKey getPostprocessorCacheKey() {
5      return new SimpleCacheKey(String.format(
6          (Locale) null,
7          "text=%s,count=%d",
8          mWatermarkText,
9          mCount));
10   }
11 }
```

Advanced: JNI and Blurring

One of the most commonly asked for post-processing effects is blurring. Luckily, Fresco ships with a very efficient implementation in native C code accessible through `NativeBlurFilter#iterativeBoxBlur`.

When you are considering more advanced post-processing, using native code is a great way to improve performance. If you go down this path, have a look at the implementation in `blur_filter.c` on how to work with bitmaps in native code. Most importantly it explains you how to lock the pixels in memory and other important tricks.

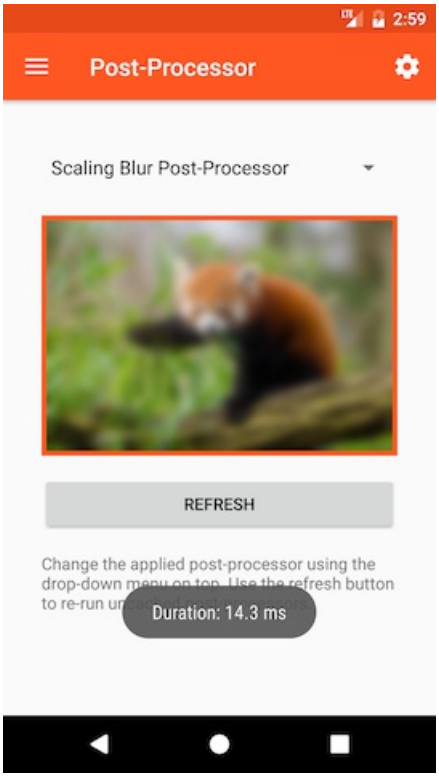


Advanced: Changing the Bitmap's Size

Even with an efficient implementation in native code, the post-processor can take a long time. For more efficient blurring, we can down-scale the image, blur the small version and then let the GPU scale it up when displayed. As blurred images do not have hard edges, this optimization usually goes unrecognized.

In our new post-processor we override an overloaded variant of the `BasePostProcessor#process()` method. That variant provides a `PlatformBitmapFactory` that we can use to create a custom output bitmap. Note that we must no longer modify the `sourceBitmap`, as it is not a copy that has been created for us.

```
public class ScalingBlurPostprocessor extends FullResolutionBlurPostprocessor {

  /**
    * A scale ration of 4 means that we reduce the total number of pixels to process by factor 16.
    */
   private static final int SCALE_RATIO = 4;

   @Override
   public CloseableReference<Bitmap> process(
       Bitmap sourceBitmap,
       PlatformBitmapFactory bitmapFactory) {
     final CloseableReference<Bitmap> bitmapRef = bitmapFactory.createBitmap(
         sourceBitmap.getWidth() / SCALE_RATIO,
         sourceBitmap.getHeight() / SCALE_RATIO);

     try {
       final Bitmap destBitmap = bitmapRef.get();
       final Canvas canvas = new Canvas(destBitmap);

       canvas.drawBitmap(
           sourceBitmap,
           null,
           new Rect(0, 0, destBitmap.getWidth(), destBitmap.getHeight()),
           mPaint);

       NativeBlurFilter.iterativeBoxBlur(destBitmap, BLUR_RADIUS / SCALE_RATIO, BLUR_ITERATIONS);

       return CloseableReference.cloneOrNull(bitmapRef);
     } finally {
       CloseableReference.closeSafely(bitmapRef);
     }
   }
}
```



Limitations

Please keep the following rules in mind when creating post-processors

- If you show the same image repeatedly, you must specify the post-processor each time it is requested. You are free to use different post-processors on different requests for the same image.
- Post-processors are not currently supported for animated images.
- If you use transparency in your post-processor, call `destinationBitmap.setHasAlpha(true);`

- Do **not** override more than one of the three `process` methods. Doing so can produce unpredictable results.
- Do **not** modify the source Bitmap when using a `process` methods that requires you to create a new destination bitmap.
- Do **not** keep a reference to either bitmap. Both have their memory managed by the image pipeline. The destBitmap will end up in your Drawee or DataSource normally.
- Do **not** use the Android `Bitmap.createBitmap` method for creating a new Bitmap. This would work against the central Bitmap pool in Fresco.

Full Sample

For the full sample see the `ImagePipelinePostProcessorFragment` in the showcase app: ImagePipelinePostProcessorFragment.java. It includes all post-processors from this page as well as additional ones.

Requesting Multiple Images (Multi-URI)

Requesting Multiple Images (Multi-URI)

The methods on this page require setting your own image request.

Going from low to high resolution

Suppose you want to show users a high-resolution, slow-to-download image. Rather than let them stare a placeholder for a while, you might want to quickly download a smaller thumbnail first.

You can set two URIs, one for the low-res image, one for the high one:

```
Uri lowResUri, highResUri;
DraweeController controller = Fresco.newDraweeControllerBuilder()
    .setLowResImageRequest(ImageRequest.fromUri(lowResUri))
    .setImageRequest(ImageRequest.fromUri(highResUri))
    .setOldController(mSimpleDraweeView.getController())
    .build();
mSimpleDraweeView.setController(controller);
```

Animated images are not supported for the low-res request.

Using thumbnail previews

This option is supported only for local URIs, and only for images in the JPEG format.

If your JPEG has a thumbnail stored in its EXIF metadata, the image pipeline can return that as an intermediate result. Your Drawee will first show the thumbnail preview, then the full image when it has finished loading and decoding.

```
 Uri uri;
 ImageRequest request = ImageRequestBuilder.newBuilderWithSource(uri)
     .setLocalThumbnailPreviewsEnabled(true)
     .build();

 DraweeController controller = Fresco.newDraweeControllerBuilder()
     .setImageRequest(request)
     .setOldController(mSimpleDraweeView.getController())
     .build();
mSimpleDraweeView.setController(controller);
```

Loading the first available image

Most of the time, an image has no more than one URI. Load it, and you're done.

But suppose you have multiple URIs for the same image. For instance, you might have uploaded an image taken from the camera. Original image would be too big to upload, so the image is downscaled first. In such case, it would be beneficial to first try to get the local-downscaled-uri, then if that fails, try to get the local-original-uri, and if even that fails, try to get the network-uploaded-uri. It would be a shame to download the image that we may have already locally.

The image pipeline normally searches for images in the memory cache first, then the disk cache, and only then goes out to the network or other source. Rather than doing this one by one for each image, we can have the pipeline check for *all* the images in the memory cache. Only if none were found would disk cache be searched in. Only if none were found there either would an external request be made.

Just create an array of image requests, and pass it to the builder.

```
Uri uri1, uri2;
ImageRequest request = ImageRequest.fromUri(uri1);
ImageRequest request2 = ImageRequest.fromUri(uri2);
ImageRequest[] requests = { request1, request2 };

DraweeController controller = Fresco.newDraweeControllerBuilder()
    .setFirstAvailableImageRequests(requests)
    .setOldController(mSimpleDraweeView.getController())
    .build();
mSimpleDraweeView.setController(controller);
```

Only one of the requests will be displayed. The first one found, whether at memory, disk, or network level, will be the one returned. The pipeline will assume the order of requests in the array is the preference order.

Specifying a custom DataSource Supplier

For even more flexibility, it is possible to specify a custom `DataSource Supplier` while building a Drawee controller. You can implement your own supplier or just compose the existing ones in whichever way you like. See `FirstAvailableDataSourceSupplier` and `IncreasingQualityDataSourceSupplier` for an example implementation. See `AbstractDraweeControllerBuilder` for how those suppliers can be composed together.

Shared Transitions

Shared Transitions

Use ChangeBounds, not ChangeImageTransform

Android 5.0 (Lollipop) introduced shared element transitions, allowing apps to share a View between multiple Activities and define a transition between them.

You can define your transitions in XML. There is a transform called ChangeImageTransform which captures an ImageView's matrix and animates it during the transition. This will not work in Fresco, which has its own set of matrices to scale with.

Fortunately there is an easy workaround. Just use the ChangeBounds transition instead. This animates the changes in the layout *bounds*. Fresco will automatically adjust the scaling matrix as you update the bounds, so your animation will appear exactly as you want it.

Using the ControllerBuilder

Using the ControllerBuilder

`SimpleDraweeView` has two methods for specifying an image. The easy way is to just call `setImageURI`.

If you want more control over how the Drawee displays your image, you can use a DraweeController. This page explains how to build and use one.

Building a DraweeController

Pass the uri to a PipelineDraweeControllerBuilder. Then specify additional options for the controller:

```
1  ControllerListener listener = new BaseControllerListener() {...}
2
3  DraweeController controller = Fresco.newDraweeControllerBuilder()
4      .setUri(uri)
5      .setTapToRetryEnabled(true)
6      .setOldController(mSimpleDraweeView.getController())
7      .setControllerListener(listener)
8      .build();
9
10 mSimpleDraweeView.setController(controller);
```

You should call `setOldController` when building a new controller. This will allow for the old controller to be reused and a couple of unnecessary memory allocations to be avoided.

More details:

- Controller Listeners

Customizing the ImageRequest

For still more advanced usage, you might need to set an ImageRequest to the pipeline, instead of merely a URI. An example of this is using a postprocessor.

```
1  Uri uri;
2  Postprocessor myPostprocessor = new Postprocessor() { ... }
3  ImageRequest request = ImageRequestBuilder.newBuilderWithSource(uri)
4      .setPostprocessor(myPostprocessor)
5      .build();
6
7  DraweeController controller = Fresco.newDraweeControllerBuilder()
8      .setImageRequest(request)
9      .setOldController(mSimpleDraweeView.getController())
10     // other setters as you need
11     .build();
```

More details:

- Postprocessors
- Requesting Multiple Images
- Resizing
- Rotation

Using the Image Pipeline Directly

Using the Image Pipeline Directly

This page is intended for advanced usage only. Most apps should be using Drawees to interact with Fresco's image pipeline.

Using the image pipeline directly is challenging because of the memory usage. Drawees automatically keep track of whether or not your images need to be in memory. They will swap them out and load them back as soon as they need to be displayed. If you are using the image pipeline directly, your app must repeat this logic.

The image pipeline returns objects wrapped in a CloseableReference. Drawees keep these references alive for as long as they need their image, and then call the `.close()` method on these references when they are finished with them. If your app is not using Drawees, it **must** do the same.

If you do not keep a Java reference to a `CloseableReference` returned by the pipleine, the `CloseableReference` will get garbage collected and the underlying `Bitmap` may get recycled while still being used. If you do not close the `CloseableReference` once you are done with it, you risk memory leaks and OOMs.

To be precise, the Java garbage collector will free image memory when Bitmap objects go out of scope, but this may be too late. Garbage collection is expensive, and relying on it for large objects leads to performance issues. This is especially true on Android 4.x and lower, when Android did not maintain a separate memory space for Bitmaps.

Calling the pipeline

You must build an image request. Having done that, you can pass it directly to the `ImagePipeline`:

```
1 ImagePipeline imagePipeline = Fresco.getImagePipeline();
2 DataSource<CloseableReference<CloseableImage>>
3     dataSource = imagePipeline.fetchDecodedImage(imageRequest, callerContext);
```

See the page on DataSources for information on how to receive data from them.

Skipping the decode

If you don't want to decode the image, but want to get the image bytes in their original compressed format, just use `fetchEncodedImage` instead:

```
1 DataSource<CloseableReference<PooledByteBuffer>>
2     dataSource = imagePipeline.fetchEncodedImage(imageRequest, callerContext);
```

Instant results from the bitmap cache

Lookups to the bitmap cache, unlike the others, are done in the UI thread. If a Bitmap is there, you get it instantly.

```
DataSource<CloseableReference<CloseableImage>> dataSource =
    imagePipeline.fetchImageFromBitmapCache(imageRequest, callerContext);
try {
  CloseableReference<CloseableImage> imageReference = dataSource.getResult();
  if (imageReference != null) {
    try {
      // Do something with the image, but do not keep the reference to it!
      // The image may get recycled as soon as the reference gets closed below.
      // If you need to keep a reference to the image, read the following sections.
    } finally {
      CloseableReference.closeSafely(imageReference);
    }
  } else {
    // cache miss
    ...
  }
} finally {
  dataSource.close();
}
```

Synchronous image loading

In a similar way to how you can immediately retrieve images from the bitmap cache, it is also possible to load an image from the network synchronously using `DataSources.waitForFinalResult()`.

```
DataSource<CloseableReference<CloseableImage>> dataSource =
    imagePipeline.fetchImageFromBitmapCache(imageRequest, callerContext);
try {
  CloseableReference<CloseableImage> result = DataSources.waitForFinalResult(dataSource);
  if (result != null) {
    // Do something with the image, but do not keep the reference to it!
    // The image may get recycled as soon as the reference gets closed below.
    // If you need to keep a reference to the image, read the following sections.
  }
} finally {
  dataSource.close();
}
```

Do **not** skip these `finally` blocks!

The caller Context

As we can see, most of the `ImagePipeline` fetch methods contains a second parameter named `callerContext` of type `Object`. We can see it as an implementation of the Context Object Design Pattern. It's basically an object we bind to a specific `ImageRequest` that can be used for different purposes (e.g. Log). The same object can also be accessed by all the `Producer` implementations into the `ImagePipeline`.

The caller Context can also be `null`.

Using Other Network Layers

Using Other Network Layers

By default, the image pipeline uses the HttpURLConnection which is included in the Android framework. However, if needed by the app a custom network layer can be used. Fresco already contains one alternative network layer that is based on OkHttp.

Using OkHttp

OkHttp is a popular open-source networking library.

1. Gradle setup

In order to use it, the `dependencies` section of your `build.gradle` file needs to be changed. Along with the Gradle dependencies given on the Getting started page, add **just one** of these:

For OkHttp2:

```
dependencies {
  // your project's other dependencies
  implementation "com.facebook.fresco:imagepipeline-okhttp:1.13.0"
}
```

For OkHttp3:

```
dependencies {
  // your project's other dependencies
  implementation "com.facebook.fresco:imagepipeline-okhttp3:1.13.0"
}
```

2. Configuring the image pipeline to use OkHttp

You must also configure the image pipeline. Instead of using `ImagePipelineConfig.newBuilder`, use `OkHttpImagePipelineConfigFactory`:

```
Context context;
OkHttpClient okHttpClient; // build on your own
ImagePipelineConfig config = OkHttpImagePipelineConfigFactory
    .newBuilder(context, okHttpClient)
    . // other setters
    . // setNetworkFetcher is already called for you
    .build();
Fresco.initialize(context, config);
```

For a more detailed example of this, see how this if configured in the Fresco showcase app.

Handling sessions and cookies correctly

The `OkHttpClient` you pass to Fresco in the above step should be set up with interceptors needed to handle authentications to your servers. See this bug and the solutions outlined there for some problems that have occurred with cookies.

Using your own network fetcher (optional)

For complete control on how the networking layer should behave, you can provide one for your app. You must subclass NetworkFetcher, which controls communications to the network. You can also optionally subclass FetchState, which is a data structure for request-specific information.

Our implementation for `OkHttp 3` can be used as an example. See its source code.

You must pass your network producer to the image pipeline when configuring it:

```
ImagePipelineConfig config = ImagePipelineConfig.newBuilder()
  .setNetworkFetcher(myNetworkFetcher);
  . // other setters
  .build();
Fresco.initialize(context, config);
```

Writing Custom Views

Writing Custom Views

DraweeHolders

There will always be times when `DraweeViews` won't fit your needs. You may need to show additional content inside the same view as your image. You might need to show multiple images inside a single view.

We provide two alternative classes you can use to host your Drawee:

- `DraweeHolder` for a single image
- `MultiDraweeHolder` for multiple images

`DraweeHolder` is a class that holds one DraweeHierarchy and the associated DraweeController. It allows you to make use of all the functionality Drawee provides in your custom views and other places where you need a drawable instead of a view. To get the drawable, you just do `mDraweeHolder.getTopLevelDrawable()`. Keep in mind that Android drawables require a bit of housekeeping which we covered below. `MultiDraweeHolder` is basically just an array of `DraweeHolder`s with some syntactic sugar added on top of it.

Responsibilities of custom views

Android lays out View objects, and only they get notified of system events. `DraweeViews` handle these events and use them to manage memory effectively. When using the holders, you must implement some of this functionality yourself.

Handling attach/detach events

Your app may leak memory, or the image may not be displayed at all, if these steps are not followed.

There is no point in images staying in memory when Android is no longer displaying the view - it may have scrolled off-screen, or otherwise not be drawing. Drawees listen for detaches and release memory when they occur. They will automatically restore the image when it comes back on-screen.

All this is automatic in a `DraweeView`, but won't happen in a custom view unless you handle four system events. These must be passed to the `DraweeHolder`. Here's how:

```
1  DraweeHolder mDraweeHolder;
2
3  @Override
4  public void onDetachedFromWindow() {
5    super.onDetachedFromWindow();
6    mDraweeHolder.onDetach();
7  }
8
9  @Override
10 public void onStartTemporaryDetach() {
11   super.onStartTemporaryDetach();
12   mDraweeHolder.onDetach();
13 }
14
15 @Override
16 public void onAttachedToWindow() {
17   super.onAttachedToWindow();
18   mDraweeHolder.onAttach();
19 }
20
21 @Override
22 public void onFinishTemporaryDetach() {
23   super.onFinishTemporaryDetach();
24   mDraweeHolder.onAttach();
25 }
```

It is important that `Holder` receives all the attach/detach events that the view itself receives. If the holder misses an attach event the image may not be displayed because Drawee will think that the view is not visible. Likewise, if the holder misses an detach event, the image may still remain in memory because Drawee will think that the view is still visible. Best way to ensure that is to create the holder from your view's constructor.

Handling touch events

If you have enabled tap to retry in your Drawee, it will not work unless you tell it that the user has touched the screen. Like this:

```
1 @Override
2 public boolean onTouchEvent(MotionEvent event) {
3   return mDraweeHolder.onTouchEvent(event) || super.onTouchEvent(event);
4 }
```

Your custom onDraw

You must call

```
1 Drawable drawable = mDraweeHolder.getTopLevelDrawable();
2 drawable.setBounds(...);
3 ...
4 drawable.draw(canvas);
```

or the Drawee won't appear at all.

- Do not downcast this Drawable. The underlying implementation may change without any notice.
- Do not translate it. Just set the proper bounds.
- If you need to apply some canvas transformations, then make sure that you properly invalidate the area that the drawable occupies in the view. See below on how to do that.

Other responsibilities

- Set Drawable.Callback

```
1 // When a holder is set to the view for the first time,
2 // don't forget to set the callback to its top-level drawable:
3 mDraweeHolder = ...
4 mDraweeHolder.getTopLevelDrawable().setCallback(this);
5
6 // In case the old holder is no longer needed,
7 // don't forget to clear the callback from its top-level drawable:
8 mDraweeHolder.getTopLevelDrawable().setCallback(null);
9 mDraweeHolder = ...
```

- Override `verifyDrawable:`

```
1 @Override
2 protected boolean verifyDrawable(Drawable who) {
3   if (who == mDraweeHolder.getTopLevelDrawable()) {
4     return true;
5   }
6   // other logic for other Drawables in your view, if any
7 }
```

- Make sure `invalidateDrawable` invalidates the region occupied by your Drawee. If you apply some canvas transformations on the drawable before it gets drawn, then those transformations needs to be taken into account in invalidation. The simplest thing to do is what Android ImageView does in its invalidateDrawable method. That is, to just invalidate the whole view when the drawable gets invalidated.

Constructing the View and DraweeHolder

This should be done carefully. See below.

Arranging your Constructors

We recommend the following pattern for constructors:

- Override all three of the three View constructors.
- Each constructor calls its superclass counterpart and then a private `init` method.
- All of your initialization happens in `init`.

That is, do not use the `this` to call one constructor from another. This is because Android View already calls one constructor from another, and it does so in an unintuitive way.

This approach guarantees that the correct initialization is called no matter what constructor is used. It is in the `init` method that your holder is created.

Creating the Holder

If possible, always create Drawees when your view gets created. Creating a hierarchy is not cheap so it's best to do it only once. More importantly, holder's lifecycle should be bound to the view's lifecycle for the reasons explained in the attach/detach section. Best way to ensure that is to create the holder when the view gets constructed as explained above.

```
1  class CustomView extends View {
2    DraweeHolder<GenericDraweeHierarchy> mDraweeHolder;
3
4    // constructors following above pattern
5
6    private void init() {
7      GenericDraweeHierarchy hierarchy = new GenericDraweeHierarchyBuilder(getResources());
8        .set...
9        .set...
10       .build();
11     mDraweeHolder = DraweeHolder.create(hierarchy, context);
12   }
13 }
```

Setting an image

Use a controller builder, but call `setController` on the holder instead of a View:

```
1 DraweeController controller = Fresco.newDraweeControllerBuilder()
2     .setUri(uri)
3     .setOldController(mDraweeHolder.getController())
4     .build();
5 mDraweeHolder.setController(controller);
```

MultiDraweeHolder

If you are dealing with multiple drawees in your custom view, `MultiDraweeHolder` might come handy. There are `add`, `remove`, and `clear` methods for dealing with DraweeHalders:

```
1  MultiDraweeHolder<GenericDraweeHierarchy> mMultiDraweeHolder;
2
3  private void init() {
4    GenericDraweeHierarchy hierarchy = new GenericDraweeHierarchyBuilder(getResources());
5      .set...
6      .build();
7    mMultiDraweeHolder = new MultiDraweeHolder<GenericDraweeHierarchy>();
8    mMultiDraweeHolder.add(new DraweeHolder<GenericDraweeHierarchy>(hierarchy, context));
9    // repeat for more hierarchies
10 }
```

You must override system events, set bounds, and do all the same responsibilities as for a single `DraweeHolder`.

Progressive JPEGs

Progressive JPEGs

Fresco supports the streaming of progressive JPEG images over the network.

Scans of the image will be shown in the view as you download them. Users will see the quality of the image start out low and gradually become clearer.

This is only supported for network images. Local images are decoded at once, so no need for progressiveness. Also, keep in mind that not all JPEG images are encoded in progressive format, and for those that are not, it is not possible to display them progressively.

Building the image request

Currently, you must explicitly request progressive rendering while building the image request:

```
1 Uri uri;
2 ImageRequest request = ImageRequestBuilder.newBuilderWithSource(uri)
3     .setProgressiveRenderingEnabled(true)
4     .build();
5 DraweeController controller = Fresco.newDraweeControllerBuilder()
6     .setImageRequest(request)
7     .setOldController(mSimpleDraweeView.getController())
8     .build();
9 mSimpleDraweeView.setController(controller);
```

We hope to add support for using progressive images with `setImageURI` in a future release.

Full Sample

For the full sample see the `ImageFormatProgressiveJpegFragment` in the showcase app:
ImageFormatProgressiveJpegFragment.java

Animated Images

Animated Images

Fresco supports animated GIF and WebP images.

We support WebP animations, even in the extended WebP format, on versions of Android going back to 2.3, even those that don't have built-in native support.

For adding this optional modules in your build.gradle please visit here:

Playing animations automatically

If you want your animated image to start playing automatically when it comes on-screen, and stop when it goes off, just say so in your image request:

```
Uri uri;
DraweeController controller = Fresco.newDraweeControllerBuilder()
    .setUri(uri)
    .setAutoPlayAnimations(true)
    . // other setters
    .build();
mSimpleDraweeView.setController(controller);
```

Playing animations manually

You may prefer to directly control the animation in your own code. In that case you'll need to listen for when the image has loaded, so it's even possible to do that.

```
 ControllerListener controllerListener = new BaseControllerListener<ImageInfo>() {
     @Override
     public void onFinalImageSet(
         String id,
         @Nullable ImageInfo imageInfo,
         @Nullable Animatable anim) {
         if (anim != null) {
           // app-specific logic to enable animation starting
           anim.start();
         }
     }
};

Uri uri;
DraweeController controller = Fresco.newDraweeControllerBuilder()
    .setUri(uri)
    .setControllerListener(controllerListener)
    // other setters
    .build();
mSimpleDraweeView.setController(controller);
```

The controller exposes an instance of the Animatable interface. If non-null, you can drive your animation with it:

```
Animatable animatable = mSimpleDraweeView.getController().getAnimatable();
if (animatable != null) {
  animatable.start();
  // later
  animatable.stop();
}
```

Limitations

Animations do not currently support postprocessors.

WebP Images

WebP Images

WebP is an image format that supports lossy and lossless compressions. Furthermore, it allows for transparency and animations.

Support on Android

Android added WebP support in version 4.0 and improved it in 4.2.1:

- 4.0+ (Ice Cream Sandwich) have basic webp support
- 4.2.1+ (Jelly Beam MR1) have support for transparency and lossless WebP

By adding the Fresco webpsupport module, apps can display all kinds of WebP images on all versions of Android:

Configuration	Basic WebP	Lossless or Transparent WebP	Animated WebP
OS < 4.0			
OS >= 4.0	✓		
OS >= 4.2.1	✓	✓	
Any OS + webpsupport	✓	✓	
Any OS + animated-webp	✓	(✓ if webpsupport or OS >= 4.2.1)	✓

Adding Support for Static WebP images on Older Versions

The only thing you need to do is add the `webpsupport` library to your dependencies. This adds support for all types of non-animated WebP images. E.g. you can use it to display transparent WebP images on Gingerbread.

```
1 dependencies {
2   // ... your app's other dependencies
3   implementation 'com.facebook.fresco:webpsupport:1.13.0'
4 }
```

Animated WebP

In order to display animated WebP images, you have to add the following dependencies:

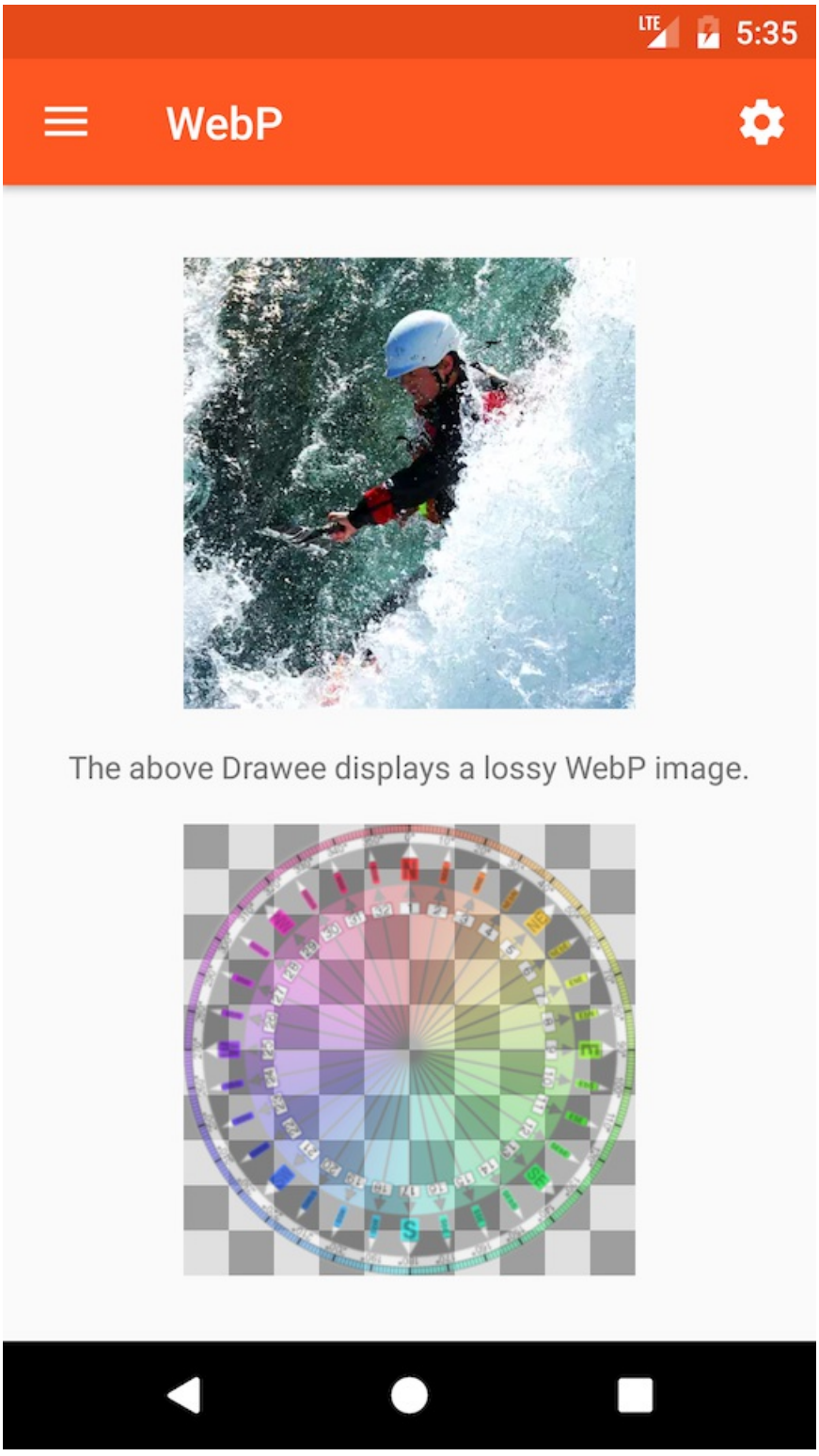
```
1 dependencies {
2   // ... your app's other dependencies
3   implementation 'com.facebook.fresco:animated-webp:1.13.0'
4   implementation 'com.facebook.fresco:webpsupport:1.13.0'
5 }
```

You can then load the animated WebP images like any other URI. In order to auto-start the animation, you can set `setAutoPlayAnimations(true)` on the `DraweeController`:

```
1 DraweeController controller = Fresco.newDraweeControllerBuilder()
2     .setUri("http://example.org/somefolder/animated.webp")
3     .setAutoPlayAnimations(true)
4     .build();
5 mSimpleDraweeView.setController(controller);
```

Full Sample

For the full sample see the `ImageFormatWebpFragment` in the showcase app: ImageFormatWebpFragment.java


5:35
WebP
The above Drawee displays a lossy WebP image.

FAQ

FAQ

These are common questions asked on our GitHub presence. Please create a pull-request if you have a Q&A that others will profit from.

How do I clear all caches?

You can use the following code to delete all cached images (both from storage and memory):

```
1 // clear both memory and disk caches
2 Fresco.getImagePipeline().clearCaches();
```

How can I create a Drawee that supports zoom gestures?

Have a look at the ZoomableDraweeView module which is part of our sample code on GitHub.

How do I create an URI for a local file?

Use the `UriUtil` class:

```
1 final File file = new File("your/file/path/img.jpg");
2 final URI uri = UriUtil.getUriForFile(file);
```

How do I create an URI for a resource?

Use the `UriUtil` class:

```
1 final int resourceId = R.drawable.my_image;
2 final URI uri = UriUtil.getUriForResourceId(resourceId);
3
4 // alternatively, if it is from another package:
5 final URI uri = UriUtil.getUriForQualifiedResource("com.myapp.plugin", resourceId);
```

How do I use Fresco in a RecyclerView?

You build your `RecyclerView` just like any other `RecyclerView`. The `DraweeView` is able to attach and detach itself appropriately. When being detached it can free up the memory of the referenced image. When being re-attached, the image is loaded from the BitmapCache if it is still available there.

Have a look at DraweeRecyclerViewFragment.java which is part of our showcase app.

How do I download a image without decoding?

For this, you can use the `imagePipeline#fetchEncodedImage(ImageRequest, ...)` method of the image pipeline. See our section on Using the Image Pipeline Directly and DataSources & DataSubscribers for detailed samples.

How do I modify an image before displaying?

The best way is to implement a PostProcessor. This allows the image pipeline to schedule the modification on the background and allocates the Bitmaps efficiently.

How large is Fresco?

If you are correctly following the steps from Shipping Your App with Fresco, your release builds should not grow more than

500 KiB when adding Fresco.

Adding support for animations (`com.facebook.fresco:animated-gif`, `com.facebook.fresco:animated-webp`) and WebP on old devices (`com.facebook.fresco:webpsupport`) is optional. This modularization allows the base Fresco library to be light-weight. Adding those additional libraries would account for ~100 KiB each.

Why can't I use Android's wrap_content attribute on a DraweeView?

The reason is that Drawee always returns -1 for getIntrinsicHeight and getIntrinsicWidth methods.

And the reason for that is that unlike a simple ImageView, Drawee may show more than one thing at the same time. For example, during the fade transition from the placeholder to the actual image, both images are visible. There may even be more than one actual image, one low-resolution, the other high-resolution. If all these images are not of exactly the same size, and they practically never are, then the concept of an "intrinsic" size cannot be well defined.

We could have returned the size of the placeholder until the image has finished loading, and then swap to the actual image's size. If we did that, though, the image would not appear correctly - it would be scaled or cropped to the placeholder's size. The only way to prevent that would be to force an Android layout pass when the image loads. Not only will that hurt your app's scroll perf, but it will be jarring for your users, who will suddenly see your app change on screen. Imagine if the user is reading a text article and all of a sudden the text jumps down because the image above it just loaded and caused everything to re-layout.

For this reason, you have to use an actual size or `match_parent` to lay out a DraweeView.

If your images are coming from a server, it may be possible to ask that server for the image dimensions, before you download it. This should be a faster request. Then use setLayoutParams to dynamically size your view upfront.

If on the other hand your use case is a legitimate exception, you can actually resize Drawee view dynamically by using a controller listener as explained here. And remember, we intentionally removed this functionality because it is undesireable. Ugly things should look ugly..

Troubleshooting

Troubleshooting

Troubleshooting

Image is displayed with repeated edges

This is a known limitation when rounding is used. See Rounding for more information and how to workaround.

Image doesn't load

You can get more information from the image pipeline by examining the verbose logcat as explained later in this section. Here are some common reasons why image loads might fail:

File not available

For example, an incorrect path for local files or an unavailable network URI is given.

Try opening a network URI in a mobile browser. If it doesn't work, the issue is likely neither in Fresco nor your app.

For a local file, try opening a file input stream directly from your app:

```
1 FileInputStream fis = new FileInputStream(new File(localUri.getPath()));
```

If that throws an exception, the issue is likely not in Fresco, **but** it may be in your app. One possibility is a permission issue, such as trying to access the SD card without requiring the necessary permission in your application manifest. Another possibility is that the pathy is not correct - perhaps you forgot to properly escape it. Finally, the file may simply not exist.

OOMs and failing to allocate a bitmap

The most common reason for this happening is loading too big images. If the image to be loaded is of considerably bigger size than the view hosting it, it should be resized.

Bitmap too large to be uploaded to a texture

Android cannot display images more than 2048 pixels long in either dimension. This is beyond the capability of the OpenGL rendering system. Fresco will resize your image if it exceeds this limit.

Investigating issues with logcat

There are various issues one might encounter when it comes to image handling. With Fresco, most of them can be diagnosed by simply looking at the `VERBOSE` logcat. This should be your starting point when investigating an issue with Fresco.

Setting up logcat

By default, Fresco does not write out all its logs. You need to configure the image pipeline to do so.

```
1 Set<RequestListener> requestListeners = new HashSet<>();
2 requestListeners.add(new RequestLoggingListener());
3 ImagePipelineConfig config = ImagePipelineConfig.newBuilder(context)
4    // other setters
5    .setRequestListeners(requestListeners)
6    .build();
7 Fresco.initialize(context, config);
8 FLog.setMinimumLoggingLevel(FLog.VERBOSE);
```

Examining logcat

All of Fresco's logs can be examined by this command:

```
1 adb logcat -v threadtime | grep -iE 'LoggingListener|AbstractDraweeController|BufferedDiskCache'
```

The output shows what is happening with the image requests within the image pipeline. It looks something like this:

```
1  08-12 09:11:14.791 6690 6690 V unknown:AbstractDraweeController: controller 28ebe0eb 0 -> 1: initialize
2  08-12 09:11:14.791 6690 6690 V unknown:AbstractDraweeController: controller 28ebe0eb 1: onDetach
3  08-12 09:11:14.791 6690 6690 V unknown:AbstractDraweeController: controller 28ebe0eb 1: setHierarchy: null
4  08-12 09:11:14.791 6690 6690 V unknown:AbstractDraweeController: controller 28ebe0eb 1: setHierarchy: com.facebook.drawee.generic.GenericDraweeHierarchy@2bb88e4
5  08-12 09:11:14.791 6690 6690 V unknown:AbstractDraweeController: controller 28ebe0eb 1: onAttach: request needs submit
6  08-12 09:11:14.791 6690 6690 V unknown:PipelineDraweeController: controller 28ebe0eb: getDataSource
7  08-12 09:11:14.791 6690 6690 V unknown:RequestLoggingListener: time 11201791: onRequestSubmit: {requestId: 1, callerContext: null, isPrefetch: false}
8  08-12 09:11:14.792 6690 6690 V unknown:RequestLoggingListener: time 11201791: onProducerStart: {requestId: 1, producer: BitmapMemoryCacheGetProducer}
9  08-12 09:11:14.792 6690 6690 V unknown:RequestLoggingListener: time 11201792: onProducerFinishWithSuccess: {requestId: 1, producer: BitmapMemoryCacheGetProducer, elapsed
10 08-12 09:11:14.792 6690 6690 V unknown:RequestLoggingListener: time 11201792: onProducerStart: {requestId: 1, producer: BackgroundThreadHandoffProducer}
11 08-12 09:11:14.792 6690 6690 V unknown:AbstractDraweeController: controller 28ebe0eb 1: submitRequest: dataSource: 36e95857
12 08-12 09:11:14.792 6690 6734 V unknown:RequestLoggingListener: time 11201792: onProducerFinishWithSuccess: {requestId: 1, producer: BackgroundThreadHandoffProducer, ela
13 08-12 09:11:14.792 6690 6734 V unknown:RequestLoggingListener: time 11201792: onProducerStart: {requestId: 1, producer: BitmapMemoryCacheProducer}
14 08-12 09:11:14.792 6690 6734 V unknown:RequestLoggingListener: time 11201792: onProducerFinishWithSuccess: {requestId: 1, producer: BitmapMemoryCacheProducer, elapsedTi
15 08-12 09:11:14.792 6690 6734 V unknown:RequestLoggingListener: time 11201792: onProducerStart: {requestId: 1, producer: EncodedMemoryCacheProducer}
16 08-12 09:11:14.792 6690 6734 V unknown:RequestLoggingListener: time 11201792: onProducerFinishWithSuccess: {requestId: 1, producer: EncodedMemoryCacheProducer, elapsedT
17 08-12 09:11:14.792 6690 6734 V unknown:RequestLoggingListener: time 11201792: onProducerStart: {requestId: 1, producer: DiskCacheProducer}
18 08-12 09:11:14.792 6690 6735 V unknown:BufferedDiskCache: Did not find image for http://www.example.com/image.jpg in staging area
19 08-12 09:11:14.793 6690 6735 V unknown:BufferedDiskCache: Disk cache read for http://www.example.com/image.jpg
20 08-12 09:11:14.793 6690 6735 V unknown:BufferedDiskCache: Disk cache miss for http://www.example.com/image.jpg
21 08-12 09:11:14.793 6690 6735 V unknown:RequestLoggingListener: time 11201793: onProducerFinishWithSuccess: {requestId: 1, producer: DiskCacheProducer, elapsedTime: 1 ms,
22 08-12 09:11:14.793 6690 6735 V unknown:RequestLoggingListener: time 11201793: onProducerStart: {requestId: 1, producer: NetworkFetchProducer}
23 08-12 09:11:15.161 6690 7358 V unknown:RequestLoggingListener: time 11202161: onProducerFinishWithSuccess: {requestId: 1, producer: NetworkFetchProducer, elapsedTime: 3
24 08-12 09:11:15.162 6690 6742 V unknown:BufferedDiskCache: About to write to disk-cache for key http://www.example.com/image.jpg
25 08-12 09:11:15.162 6690 6734 V unknown:RequestLoggingListener: time 11202162: onProducerStart: {requestId: 1, producer: DecodeProducer}
26 08-12 09:11:15.163 6690 6742 V unknown:BufferedDiskCache: Successful disk-cache write for key http://www.example.com/image.jpg
27 08-12 09:11:15.169 6690 6734 V unknown:RequestLoggingListener: time 11202169: onProducerFinishWithSuccess: {requestId: 1, producer: DecodeProducer, elapsedTime: 7 ms, e
28 08-12 09:11:15.169 6690 6734 V unknown:RequestLoggingListener: time 11202169: onRequestSuccess: {requestId: 1, elapsedTime: 378 ms}
29 08-12 09:11:15.184 6690 6690 V unknown:AbstractDraweeController: controller 28ebe0eb 1: set_final_result @ onNewResult: image: CloseableReference 2fd41bb0
```

In this case, we see that the controller `28ebe0eb` associated with a DraweeView started datasource `36e95857` which issued image request `1`. We can now see that the image was not found in the bitmap cache, nor in the encoded memory cache, nor in the disk cache, and so the network fetch had to be performed. The fetch was successful, the image was decoded and the request finished successfully. Finally, the datasource notified the controller which then set the resulting image to the hierarchy (`set_final_result`).

Gotchas

Gotchas

Don't use ScrollViews

If you want to scroll through a long list of images, you should use a RecyclerView, ListView, or GridView. All of these re-use their child views continually as you scroll through them. Fresco descendant views receive the system events that let them manage memory correctly.

`ScrollView` does not do this. Thus, Fresco views aren't told when they have gone off-screen, and hold onto their image memory until your Fragment or Activity is stopped. Your app will be at a much greater risk of OOMs.

Don't downcast

It is tempting to downcast objects returned by Fresco classes into actual objects that appear to give you greater control. At best, this will result in fragile code that gets broken in next release; at worst, it will lead to very subtle bugs.

Don't use getTopLevelDrawable

`DraweeHierarchy.getTopLevelDrawable()` should **only** be used by DraweeViews. Client code should almost never interact with it.

The sole exception is custom views. Even there, the top-level drawable should never be downcast. We may change the actual type of the drawable in future releases.

Don't re-use DraweeHierarchies

Never call `DraweeView.setHierarchy` with the same argument on two different views. Hierarchies are made up of Drawables, and Drawables on Android cannot be shared among multiple views.

Re-use Drawable resource IDs, not Java Drawable objects

This is for the same reason as the above. Drawables cannot be shared in multiple views.

You can freely use the same `@drawable` resource ID as a placeholder, error, or retry in multiple `SimpleDraweeViews` in XML. If you are using `GenericDraweeHierarchyBuilder`, you must call Resources.getDrawable separate for *each* hierarchy. Do not call it just once and pass it to multiple hierarchies!

Do not control hierarchy directly

Do not interact with `SettableDraweeHierarchy` methods (`reset`, `setImage`, …). Those are to be used by controller only. Do NOT be tempted to use `setControllerOverlay` in order to set an overlay. This method is to be called by controller only, and it refers to a very special controller overaly. If you just need to display an overlay see [Drawee branches] (http://frescolib.org/docs/drawee-branches.html#Overlays).

Don't set images directly on a DraweeView

Currently `DraweeView` is a subclass of Android's ImageView. This has various methods to set an image (such as setImageBitmap, setImageDrawable)

If you set an image directly, you will completely lose your `DraweeHierarchy`, and will not get any results from the image pipeline.

Don't use ImageView attributes or methods with DraweeView

Any XML attribute or method of ImageView not found in View will not work on a DraweeView. Typical cases are `src`, `scaleType`, `adjustViewBounds`, etc. Don't use those. DraweeView has its own counterparts as explained in the other sections of

this documentation. Any ImageView attrribute or method will be removed in the upcoming release, so please don’t use those.

Building from Source

Building from Source

You should only build from source if you need to modify Fresco code itself. Most applications should simply include Fresco in their project.

Prerequisites

The following tools must be installed on your system in order to build Fresco:

1. The Android SDK
2. From the Android SDK Manager, install/upgrade the latest Support Library **and** Support Repository. Both are found in the Extras section.
3. The Android NDK. Version 10c or later is required.
4. The git version control system.

You don't need to download Gradle itself; the build scripts or Android Studio will do that for you.

Fresco does not support source builds with Eclipse, Ant, or Maven. We do not plan to ever add such support.

Configuring Gradle

Both command-line and Android Studio users need to edit the `gradle.properties` file. This is normally located in your home directory, in a subdirectory called `.gradle`. If it is not already there, create it.

On Unix-like systems, including Mac OS X, add this line:

```
1 ndk.path=/path/to/android_ndk/r10e
```

On Windows systems, add this line:

```
1 ndk.path=C\:\\path\\to\\android_ndk\\r10e
```

On *both* platforms, add these lines:

```
1 org.gradle.daemon=true
2 org.gradle.parallel=true
3 org.gradle.configureondemand=true
```

Windows' backslashes and colons need to be escaped in order for Gradle to read them correctly.

Getting the source

```
1 git clone https://github.com/facebook/fresco.git
```

This will create a directory `fresco` where the code will live.

Building from the Command Line

On Unix-like systems, `cd` to the directory containing Fresco. Run the following command:

```
1 ./gradlew build
```

On Windows, open a Command Prompt, `cd` to the directory containing Fresco, and type in this command:

```
gradlew.bat build
```

Building from Android Studio

From Android Studio's Quick Start dialog, click Import Project. Navigate to the directory containing Fresco and click on the `build.gradle` file.

Android Studio should build Fresco automatically.

Offline builds

The first time you build Fresco, your computer must be connected to the Internet. Incremental builds can use Gradle's `--offline` option.

Troubleshooting

> Could not find com.android.support:…:x.x.x.

Make sure your Support Repository is up to date (see Prerequisites above).

Windows support

We try our best to support building on Windows but we can't commit to it. We do not have a Windows build set up on our CI servers and none of us is using a Windows computer so the builds can break without us noticing it.

Please raise github issues if the Windows build is broken or submit a pull request with the fix. We do our best but we'd like the community's help to keep this up to date.

Contributing code upstream

Please see our CONTRIBUTING page.

Sample code

Sample code

Note: the samples are licensed for non-commercial or evaluation purposes only, not the MIT license used for Fresco itself.

Fresco's GitHub repository contains several samples to demonstrate how to use Fresco in your apps.

The samples are available in source form only. Follow the build instructions to set up your dev environment to build and run them.

The Showcase app

The Showcase App demonstrates various features and allows to customize parameters to show their effect. It includes samples for Drawee and for the image pipeline. Furthermore, it showcases how to use both built-in and custom image formats.

The zoomable library

The zoomable library features a `ZoomableDraweeView` class that supports gestures such as pinch-to-zoom and panning of a Drawee image.

The comparison app

The comparison app lets the user do a proper, apples-to-apples comparison of Fresco with Picasso, Universal Image Loader, Volley's image loader, and Glide.

Fresco allows you to also compare its performance with OkHttp as its network layer. You can also see the performance of Drawee running over Volley instead of Fresco's image pipeline.

The app offers you a choice of images from your local camera or from the Internet. The network images come from Imgur.

You can build, install, and run a controlled test of any combination of loaders using the run_comparison.py script. The following command will run them all on a connected ARM v7 device:

```
./run_comparison.py -c armeabi-v7a
```

The round app

The round app shows the same image scaled in several different ways, with and without a circle applied.

Concepts

Concepts

Drawees

Drawees are spaces in which images are rendered. These are made up of three components, like a Model-View-Controller (MVC) framework.

DraweeView

Descended from the Android View class.

Most apps should use the `SimpleDraweeView` class. Place these in your application using XML or Java code. Set the URI to load with the `setImageURI` method, as explained in the Getting Started page.

See Using SimpleDraweeView.

DraweeHierarchy

This is the hierarchy of Android Drawable objects that will actually render your content. Think of it as the Model in an MVC.

See Using SimpleDraweeView.

DraweeController

The `DraweeController` is the class responsible for actually dealing with the underlying image loader - whether Fresco's own image pipeline, or another.

If you need something more than a single URI to specify the image you want to display, you will need an instance of this class.

DraweeControllerBuilder

`DraweeControllers` are immutable once constructed. They are built using the Builder pattern.

Listeners

One use of a builder is to specify a Listener to execute code upon the arrival, full or partial, of image data from the server.

The Image Pipeline

Behind the scenes, Fresco's image pipeline deals with the work done in getting an image. It fetches from the network, a local file, a content provider, or a local resource. It keeps a cache of compressed images on local storage, and a second cache of decompressed images in memory.

The image pipeline uses a special technique called *pinned purgeables* to keep images off the Java heap. This requires callers to `close` images when they are done with them.

`SimpleDraweeView` does this for you automatically, so should be your first choice. Very few apps need to use the image pipeline directly.

Drawee Branches

Drawee Branches

What are Branches?

Drawees are made up of different image “branches”, one or more of which may be actually displayed at a time.

This page outlines the different branches that can be displayed in a Drawee, and how they are set.

Except for the actual image, all of them can be set by an XML attribute. The value in XML must be either an Android drawable or color resource.

They can also be set by a method in the GenericDraweeHierarchyBuilder class, if setting programmatically. In code, the value can either be from resources or be a custom subclass of Drawable.

Some of the items can even be changed on the fly after the hierarchy has been built. These have a method in the GenericDraweeHierarchy class.

Several of the drawables can be scaled.

Actual

The *actual* image is the target; everything else is either an alternative or a decoration. This is specified using a URI, which can point to an image over the Internet, a local file, a resource, or a content provider.

This is a property of the controller, not the hierarchy. It therefore is not set by any of the methods used by the other Drawee branches.

Instead, use the `setImageURI` method or set a controller programmatically.

In addition to the scale type, the hierarchy exposes other methods only for the actual image. These are:

- the focus point (used for the focusCrop scale type only)
- a color filter

Default scale type: `centerCrop`

Placeholder

The *placeholder* is shown in the Drawee when it first appears on screen. After you have called `setController` or `setImageURI` to load an image, the placeholder continues to be shown until the image has loaded.

In the case of a progressive JPEG, the placeholder only stays until your image has reached the quality threshold, whether the default, or one set by your app.

XML attribute: `placeholderImage` Hierarchy builder method: `setPlaceholderImage` Hierarchy mutation method: `setPlaceholderImage` Default value: None Default scale type: `centerInside`

Failure

The *failure* image appears if there is an error loading your image. The most common cause of this is an invalid URI, or lack of connection to the network.

XML attribute: `failureImage` Hierarchy builder method: `setFailureImage` Hierarchy mutation method: `setFailureImage` Default value: None Default scale type: `centerInside`

Retry

The *retry* image appears instead of the failure image if you have set your controller to enable the tap-to-retry feature.

You must build your own Controller to do this. Then add the following line

```
1 .setTapToRetryEnabled(true)
```

The image pipeline will then attempt to retry an image if the user taps on it. Up to four attempts are allowed before the failure image is shown instead.

XML attribute: `retryImage` Hierarchy builder method: `setRetryImage` Hierarchy mutation method: `setRetryImage` Default value: None Default scale type: `centerInside`

Progress Bar

If specified, the *progress bar* image is shown as an overlay over the Drawee until the final image is set.

For more details, see the progress bar page.

XML attribute: `progressBarImage` Hierarchy builder method: `setProgressBarImage` Hierarchy mutation method: `setProgressBarImage` Default value: None Default scale type: `centerInside`

Backgrounds

Background drawables are drawn first, “under” the rest of the hierarchy.

Only one can be specified in XML, but in code more than one can be set. In that case, the first one in the list is drawn first, at the bottom.

Background images don’t support scale-types and are scaled to the Drawee size.

XML attribute: `backgroundImage` Hierarchy builder method: `setBackground, setBackgrounds` Default value: None Default scale type: N/A

Overlays

Overlay drawables are drawn last, “over” the rest of the hierarchy.

Only one can be specified in XML, but in code more than one can be set. In that case, the first one in the list is drawn first, at the bottom.

Overlay images don’t support scale-types and are scaled to the Drawee size.

XML attribute: `overlayImage` Hierarchy builder method: `setOverlay, setOverlays` Default value: None Default scale type: N/A

Pressed State Overlay

The *pressed state overlay* is a special overlay shown only when the user presses the screen area of the Drawee. For example, if the Drawee is showing a button, this overlay could have the button change color when pressed.

The pressed state overlay doesn’t support scale-types.

XML attribute: `pressedStateOverlayImage` Hierarchy builder method: `setPressedStateOverlay` Default value: None Default scale type: N/A

Introduction to the Image Pipeline

Introduction to the Image Pipeline

The image pipeline does everything necessary to get an image into a form where it can be rendered into an Android device.

The pipeline goes through the following steps when given an image to load:

1. Look in the bitmap cache. If found, return it.
2. Hand off to other threads.
3. Check in the encoded memory cache. If found, decode, transform, and return it. Store in the bitmap cache.
4. Check in the disk cache. If found, decode, transform, and return it. Store in the encoded-memory and bitmap caches.
5. Check on the network (or other original source). If found, decode, transform, and return it. Store in all three caches.

This being an image library, an image is worth a thousand words:

(The ‘disk’ cache as pictured includes the encoded memory cache, to keep the logic path clearer.) See this page for more details on caching.

The pipeline can read from local files as well as network. PNG, GIF, and WebP are supported as well as JPEG.