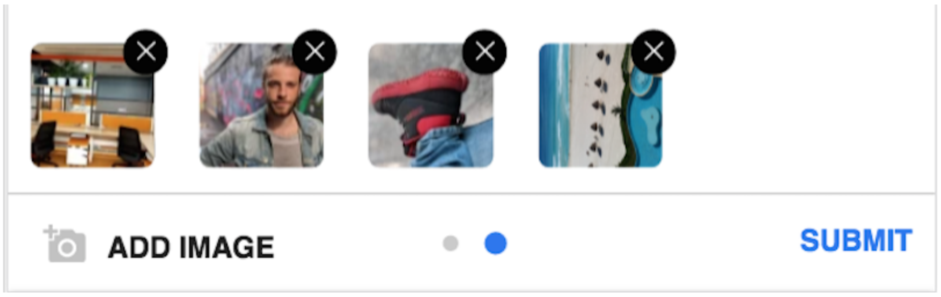
Use case is pretty simple. User should be able to upload images (either from gallery or use the camera realtime) while writing review about a product. Also the user should be able to view images in reviews for a product. This email will mostly focus on learnings from the former flow, i.e. uploading images.

User when uploads an image, the app should show a thumbnail (preview) of the uploaded file. And also the image should be, of course, uploaded to a file service.

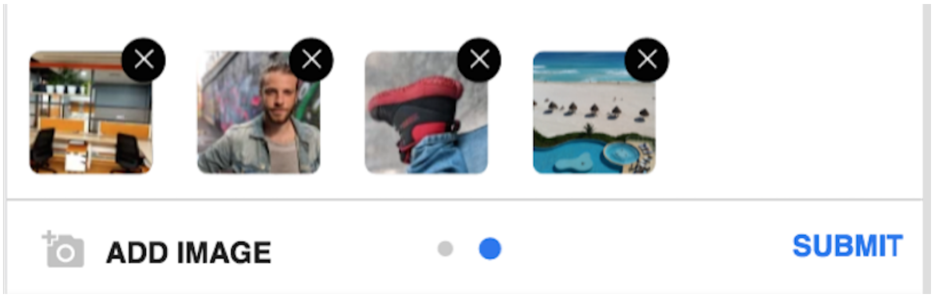
Let's talk about showing preview images. To show image preview, we need to read the file uploaded and prepare a URL or a base64 encoded string as the src of the preview image element. This is simple indeed. However, there's one big challenge here. Since the image can be uploaded from your gallery or say camera, the image can have a lot of metadata which could potentially impact how you see the image. Such metadata is coded into something called EXIF data like shutter speed, aperture, phone make/model GPS location during the shoot, orientation etc. What's important for us is to understand the orientation flag in the EXIF data. Why ?? Because not every software auto-rotates the photo while viewing depending on the orientation of the camera. Browsers do NOT handle natively the orientation of the image. Like, a portrait image could be modelled as a landscape rotated by -90 degrees.

<u>Extracting EXIF orientation</u>

To make this work, we had to read through the EXIF orientation flag to determine what is the potential orientation and then rotate it manually to show the images with the correct orientation. Now, calculating this flag from the EXIF data, it's quite a computationally expensive operation. So we spawn a worker to compute that EXIF information and let the main thread know the orientation of the image. Take a look at the 4th image in the screenshots below. The first one is without exif adjustment while the latter is with.



(without exif orientation)



(with exif orientation)

Now that we have the orientation of the image, we need to rotate the image (or rather image data). This is done via spreading the image over a canvas, then rotate the canvas information and then getting the data back. What's important to consider here is that, technically we can either use:

1. **toDataURL** - generates a base64 encoded string for the canvas
2. **toBlob** - a blob representing the canvas data

for getting the URL which can be used to set against an **<img />** element. Assessment of these two methods brought substantial differences:

toDataURL	toBlob
Sync	Async
compressed to JPG/PNG and then converted to string. Quite larger in size.	compressed to JPG/PNG. Smaller size close to 30%.
Quite slow to execute. Close to 300-400ms reduction.	Faster to execute

So based on these numbers, we decided to use toBlob instead of toDataURL.

Optimising Worker communication

Another area of optimisation was for the Worker instance that computed the orientation from the imageArrayBuffer. Since the postMessage clones objects using the Structured Clone Algorithm, it's still an expensive operation because it's a copy operation in the end. So, to reduce the potential copying time, we use Transferables which can be used to transfer the ArrayBuffer from the main thread to the worker context instead of copying it. The following shows the impact of using transferables:

Image size **1.7MB**. The values in the table are in *ms*.

Clone	Transferables
65	11
74	14
52	12

So, our implementation uses Transferables if supported else uses cloning as a fallback.

Resampling uploaded images

Now since the uploaded image weighs around 2MB (max size), we would work it to reduce the payload size to save data on the client. Image resampling is what was needed to be done. Image resampling does modify the image using constructs called "filters" which are applied to a given image to generate another. Accordiing to ImageMagick, the way filers work is:

When resizing an image you are basically trying to determine the correct value of each pixel in the new image, based on the pixels in the original source image. However these new pixels do not match exactly to the positions of the old pixels, and so a correct value for these pixels needs to be determined in some way.
What is done is to try to use some type of weighted average of the original source pixel values to determine a good value for the new pixel. The real pixels surrounding the location of the new pixel forms a 'neighbourhood' of contributing values. The larger this neighbourhood is the slower the resize. This is a technique called Convolution.
For the sake of simplicity and limited knowledge in the domain, we chose to use the simplest type of filters i.e. Interpolation filters.

These take a specific pixel location in the source image and try to simply determine a logical color value of the image at that location based on the colors of the surrounding pixels.

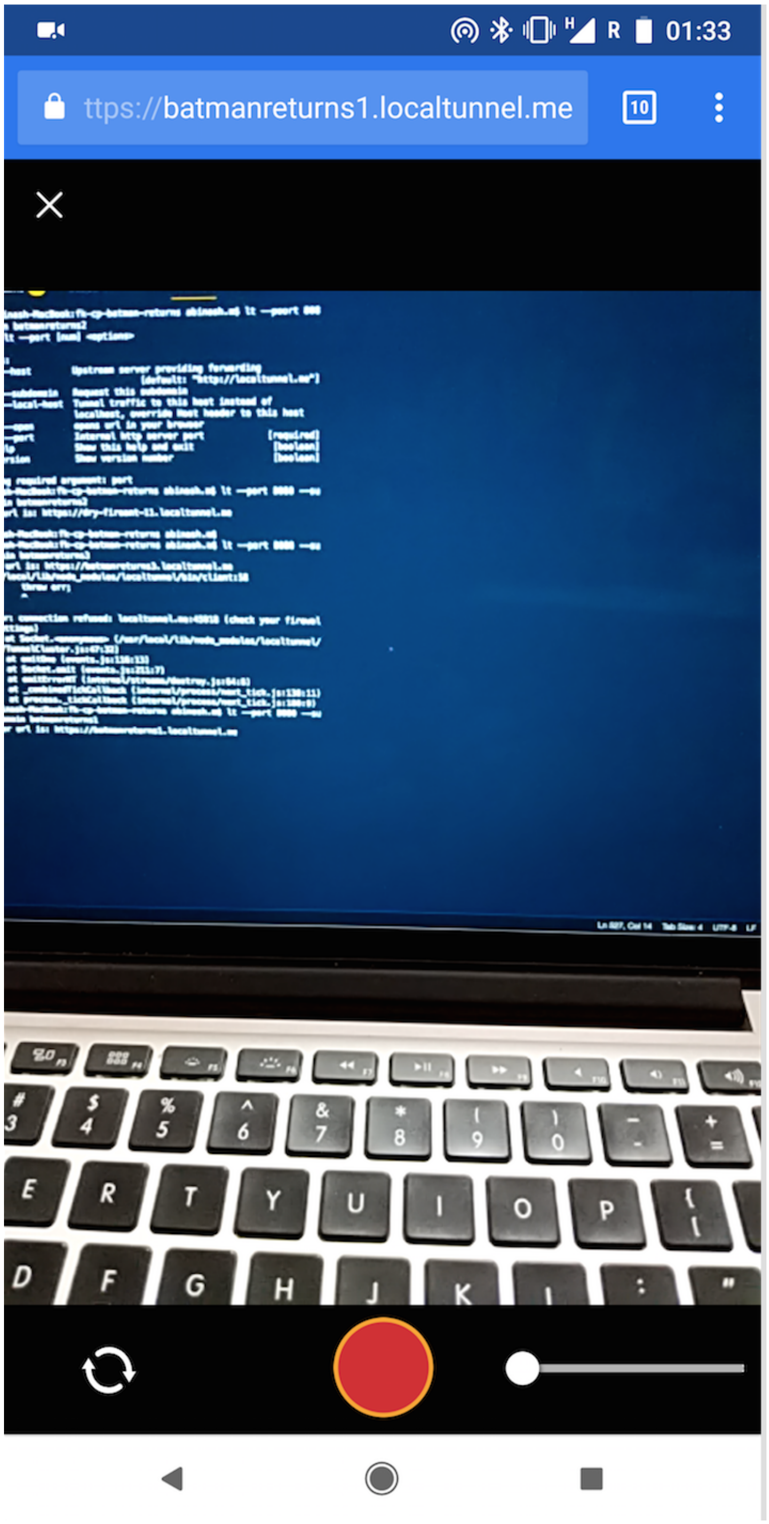
The type of interpolation filter we used is the **Hermite** interpolation filter. There are other interpolation filters available like Lagrange, Catrom etc. Advantage for Hermite is that it produces a smooth interpolation of the image data.
For our case, we resize and resample our initial image with dimensions: W and H, to dimensions: W / 2 and H / 2 and run Hermite resample. This gives considerable reduction in the file/blob size. We spread the image over a canvas and run the resize and resample. We also tinker with the quality factor while finally generating the blob from the canvas. Basically, the **HTMLCanvasElement.toBlob()** method also takes in the mime-type and output quality for the generated blob. The default value for the quality is 0.92 (in most major browsers), while we are generating the blobs at 0.97.
We did experiment with **[1, 0.97, 0.95, 0.9, 0.85, 0.8]**. When the quality factor was 1, we found for some images, the size was exorbitantly high. Reducing from 1 reduced the size considerably. But no "significant" changes in the quality for other values. So we've settled for value 0.97 as quality factor.

https://rukminim1.flixcart.com/blobio/1000/1000/20180507/blobio-20180507_tvq28tn8.jpeg?q=90 (quality = 1)
https://rukminim1.flixcart.com/blobio/1000/1000/20180507/blobio-20180507_svk325lp.jpeg?q=90 (quality = 0.8)

Camera and the web

Since the other way of uploading images was from the device camera, we built a small virtual camera which streams the input from the device camera. It hosts the capabilities of:

1. streaming the device camera feed
2. rear/front camera flip
3. Image zoom
4. Take a photo (image capture)



```
inesh-MacBook:fk-cp-batman-returns abinesh.m$ lt --port 808
n batmanreturns2
lt --port [num] <options>

:
-host          Upstream server providing forwarding
                          (default: "http://localtunnel.me")
--subdomain    Request this subdomain
--local-host   Tunnel traffic to this host instead of
               localhost, override Host header to this host
--open         opens url in your browser
--port         Internal http server port              [required]
lp             Show this help and exit                [boolean]
rsion          Show version number                    [boolean]

g required argument: port
h-MacBook:fk-cp-batman-returns abinesh.m$ lt --port 8080 --su
in batmanreturns2
rl is: https://dry-fireant-11.localtunnel.me

sh-MacBook:fk-cp-batman-returns abinesh.m$
sh-MacBook:fk-cp-batman-returns abinesh.m$ lt --port 8080 --su
in batmanreturns3
url is: https://batmanreturns3.localtunnel.me
/local/lib/node_modules/localtunnel/bin/client:58
    throw err;
    ^

r: connection refused: localtunnel.me:45018 (check your firewal
ttings)
 at Socket.<anonymous> (/usr/local/lib/node_modules/localtunnel/
TunnelCluster.js:47:32)
 at emitOne (events.js:116:13)
 at Socket.emit (events.js:211:7)
 at emitErrorNT (internal/streams/destroy.js:64:8)
 at _combinedTickCallback (internal/process/next_tick.js:138:11)
 at process._tickCallback (internal/process/next_tick.js:180:9)
inesh-MacBook:fk-cp-batman-returns abinesh.m$ lt --port 8080 --su
main batmanreturns1
r url is: https://batmanreturns1.localtunnel.me
```

This is powered using WebRTC. WebRTC now allows to read more detailed information about camera settings like the zoom levels of the device camera, constraints like ISO, brightness, contrast etc.