Take Photos and Control Camera Settings

By Miguel Casas-Sanchez

Miguel is an engineer in the Chrome team

By Sam Dutton

Sam is a Developer Advocate

By François Beaufort

Dives into Chromium source code

Image Capture is an API to capture still images and configure camera hardware settings. This API is available in Chrome 59 on Android and desktop. We've also published an ImageCapture polyfill library.

The API enables control over camera features such as zoom, brightness, contrast, ISO and white balance. Best of all, Image Capture allows you to access the full resolution capabilities of any available device camera or webcam. Previous techniques for taking photos on the Web have used video snapshots, which are lower resolution than that available for still images.

An **`ImageCapture`** object is constructed with a **`MediaStreamTrack`** as source. The API has then two capture methods **`takePhoto()`** and **`grabFrame()`** and ways to retrieve the capabilities and settings of the camera, and to change those settings.

Construction

The Image Capture API gets access to a camera via a **`MediaStreamTrack`** obtained from **`getUserMedia()`**:

```
navigator.mediaDevices.getUserMedia({video: true})
  .then(gotMedia)
  .catch(error => console.error('getUserMedia() error:', error));

function gotMedia(mediaStream) {
  const mediaStreamTrack = mediaStream.getVideoTracks()[0];
  const imageCapture = new ImageCapture(mediaStreamTrack);
  console.log(imageCapture);
}
```

You can try out this code from the DevTools console.

Note: To choose between different cameras, such as the front and back camera on a phone, get a list of available devices via the `navigator.mediaDevices.enumerateDevices()` method, then set `deviceId` in `getUserMedia()` constraints as per the demo here.

Capture

Capture can be done in two ways: full frame and quick snapshot. `takePhoto()` returns a `Blob`, the result of a single photographic exposure, which can be downloaded, stored by the browser or displayed in an `<img>` element. This method uses the highest available photographic camera resolution. For example:

```
const img = document.querySelector('img');
// ...
imageCapture.takePhoto()
  .then(blob => {
    img.src = URL.createObjectURL(blob);
    img.onload = () => { URL.revokeObjectURL(this.src); }
  })
  .catch(error => console.error('takePhoto() error:', error));
```

`grabFrame()` returns an `ImageBitmap` object, a snapshot of live video, which could (for example) be drawn on a `<canvas>` and then post-processed to selectively change color values. Note that the `ImageBitmap` will only have the resolution of the video source — which will usually be lower than the camera's still-image capabilities. For example:

```
const canvas = document.querySelector('canvas');
// ...
imageCapture.grabFrame()
  .then(imageBitmap => {
    canvas.width = imageBitmap.width;
    canvas.height = imageBitmap.height;
    canvas.getContext('2d').drawImage(imageBitmap, 0, 0);
  })
  .catch(error => console.error('grabFrame() error:', error));
```

Capabilities and settings

There are a number of ways to manipulate the capture settings, depending on whether the changes would be reflected in the **`MediaStreamTrack`** or can only be seen after **`takePhoto()`**. For example, a change in **`zoom`** level is immediately propagated to the **`MediaStreamTrack`** whereas the red eye reduction, when set, is only applied when the photo is being taken.

"Live" camera capabilities and settings are manipulated via the preview **`MediaStreamTrack`**: **`MediaStreamTrack.getCapabilities()`** returns a **`MediaTrackCapabilities`** dictionary with the concrete supported capabilities and the ranges or allowed values, e.g. supported zoom range or allowed white balance modes. Correspondingly, **`MediaStreamTrack.getSettings()`** returns a **`MediaTrackSettings`** with the concrete current settings. Zoom, brightness, and torch mode belong to this category, for example:

```
var zoomSlider = document.querySelector('input[type=range]');
// ...
const capabilities = mediaStreamTrack.getCapabilities();
const settings = mediaStreamTrack.getSettings();
if (capabilities.zoom) {
  zoomSlider.min = capabilities.zoom.min;
  zoomSlider.max = capabilities.zoom.max;
  zoomSlider.step = capabilities.zoom.step;
  zoomSlider.value = settings.zoom;
}
```

Note: **`MediaStreamTrack.getCapabilities()`** and **`.getSettings()`** are synchronous but the capabilities are only available when the camera starts actually streaming, so make sure there is a delay between calling these methods and **`getUserMedia()`**, see crbug.com/711524.

"Non-Live" camera capabilities and settings are manipulated via the **`ImageCapture`** object: **`ImageCapture.getPhotoCapabilities()`** returns a **`PhotoCapabilities`** object that provides access to "Non-Live" available camera capabilities. Correspondingly, starting in Chrome 61, **`ImageCapture.getPhotoSettings()`** returns a **`PhotoSettings`** object with the concrete current settings. The photo resolution, red eye reduction and flash mode (except torch) belong to this section, for example:

```
var widthSlider = document.querySelector('input[type=range]');
// ...
imageCapture.getPhotoCapabilities()
  .then(function(photoCapabilities) {
    widthSlider.min = photoCapabilities.imageWidth.min;
    widthSlider.max = photoCapabilities.imageWidth.max;
    widthSlider.step = photoCapabilities.imageWidth.step;
    return imageCapture.getPhotoSettings();
  })
```

```
  .then(function(photoSettings) {
    widthSlider.value = photoSettings.imageWidth;
  })
  .catch(error => console.error('Error getting camera capabilities and settings:'
```

Configuring

"Live" camera settings can be configured via the preview **MediaStreamTrack**'s **applyConstraints()** constraints , for example:

```
var zoomSlider = document.querySelector('input[type=range]');

mediaStreamTrack.applyConstraints({ advanced: [{ zoom: zoomSlider.value }]})
  .catch(error => console.error('Uh, oh, applyConstraints() error:', error));
```

"Non-Live" camera settings are configured with the **takePhoto()**'s optional **PhotoSettings** dictionary, for example:

```
var widthSlider = document.querySelector('input[type=range]');
imageCapture.takePhoto({ imageWidth : widthSlider.value })
  .then(blob => {
    img.src = URL.createObjectURL(blob);
    img.onload = () => { URL.revokeObjectURL(this.src); }
  })
  .catch(error => console.error('Uh, oh, takePhoto() error:', error));
```

Camera capabilities

If you run the code above, you'll notice a difference in dimensions between the **grabFrame()** and **takePhoto()** results.

The **takePhoto()** method gives access to the camera's maximum resolution.

grabFrame() just takes the next available **VideoFrame** in the **MediaStreamTrack** inside the renderer process, whereas **takePhoto()** interrupts the **MediaStream**, reconfigures the camera, takes the photo (usually in a compressed format, hence the **Blob**) and then resumes the **MediaStreamTrack**. In essence, this means that **takePhoto()** gives access to the full still-image resolution capabilities of the camera. Previously, it was only possible to 'take a photo' by calling **drawImage()** on a **canvas** element, using a video as the source (as per the example here).

More information can be found in the README.md section.

In this demo, the `<canvas>` dimensions are set to the resolution of the video stream, whereas the natural size of the `<img>` is the maximum still-image resolution of the camera. CSS, of course, is used to set the display size of both.

The full range of available camera resolutions for still images can be get and set using the `MediaSettingsRange` values for `PhotoCapabilities.imageHeight` and `imageWidth`. Note that the minimum and maximum width and height constraints for `getUserMedia()` are for video, which (as discussed) may be different from the camera capabilities for still images. In other words, you may not be able to access the full resolution capabilities of your device when saving from `getUserMedia()` to a canvas. The WebRTC resolution constraint demo shows how to set `getUserMedia()` constraints for resolution.

Anything else?

- The **Shape Detection API** works well with Image Capture: `grabFrame()` can be called repeatedly to feed `ImageBitmaps` to a `FaceDetector` or `BarcodeDetector`. Find out more about the API from Paul Kinlan's blog post.
- The **Camera flash** (device light) can be accessed via `FillLightMode` in `PhotoCapabilities` , but the **Torch mode** (flash constantly on) can be found in the `MediaTrackCapabilities`.

Demos and code samples

- Chrome Samples demo
- simpl.info/ic
- WebRTC samples

Support

- Chrome 59 on Android and desktop.
- Chrome Canary on Android and desktop previous to 59 with **Experimental Web Platform** features enabled.

Find out more

- W3C Image Capture Spec
- Image Capture implementation status
- Shape Detection API
- Shape Detection explainer and readme
- Shape Detection demo collection

Last updated July 2, 2018.