Using the Browser's <canvas> for Data Compression

When building static websites and Single-Page Applications (SPAs), we sometimes need functionality in JavaScript front ends—such as compression—that is usually handled on the back end instead.[1] For example, to store SPA state in the URL hash (the part after the #, also known as the fragment), we want the serialized data to be as small as possible.[2] In such cases, we would benefit from accessing browsers' compression implementations.[3]

Web browsers typically include optimized data compression libraries because they compress and decompress HTTP requests and images, among other data types.[4,5] Yet data compression APIs were not widely accessible from websites' JavaScript front ends until May 2023.[6]

Most modern browsers have implemented the Compression Streams API, thereby supporting compression directly from JavaScript.[7] But how do we use compression functionality in old browsers where it is not exposed? It turns out that it is not *directly* exposed, but is *indirectly* exposed: if we can put data into a format that is compressed by the browser, and then get the resulting file, then that file will contain a compressed version of our data. Specifically, we can compress arbitrary data by leveraging browsers' ability to losslessly compress pixel data into a PNG. Even accounting for headers, checksums, and overhead from the PNG format, the resulting file is usually smaller than the uncompressed data.

```
// Uint8Array -> compressed base64 string
function compress(data) {
  data = Array.from(data);
  // Last pixel can have 1-3 data bytes. Store
  // that number in the first byte
  data.unshift(data.length % 3);
  const c = document.createElement("canvas");
  const numPixels = Math.ceil(data.length / 3);
  c.width = numPixels;
  c.height = 1;
  const context = c.getContext("2d");
  context.fillStyle = "white";
  context.fillRect(0, 0, c.width, c.height);
  const image = context.getImageData(
    0, 0, c.width, c.height,
  );
  let offset = 0;
  for (const b of data) {
    // The alpha channel must be fully opaque or
    // there will be cross-browser inconsistencies
    // when encoding and decoding pixel data
    if (offset % 4 == 3) {
      image.data[offset++] = 255;
    }
    image.data[offset++] = b;
  }
  context.putImageData(image, 0, 0);
  const url = c.toDataURL("image/png");
  return url.match(/,(.*)/)[1];
}
```

```
// compressed base64 string -> original Uint8Array
function decompress(base64) {
  // Decompression must be async. There is a race
  // if we don't wait for the image to load before
  // using its pixels
  return new Promise((resolve, reject) => {
    const img = document.createElement("img");
    img.onerror = () => reject(
      new Error("Could not extract image data")
    );
    img.onload = () => {
      try {
        const c =
          document.createElement("canvas");
        c.width = img.naturalWidth;
        c.height = img.naturalHeight;
        const context = c.getContext("2d");
        context.drawImage(img, 0, 0);
        const raw = context.getImageData(
          0, 0, c.width, c.height,
        ).data;
        // Filter out the alpha channel
        const r = raw.filter((_, i) => i%4 != 3);
        resolve(new Uint8Array(
          r.slice(1, r.length - 3 + r[0] + 1),
        ));
      } catch (e) { reject(e); }
    };
    img.src = `${base64}`;
  });
}
```

[1] Another example is base64-encoding arbitrary byte sequences. JavaScript has `btoa` and `atob` for converting strings to and from base64 encoding, but those functions fail for byte sequences that are not valid UTF-16 strings. In other words, they don't work on all `Uint8Arrays` and, therefore, cannot encode or decode truly arbitrary byte sequences.

[2] Browsers have varying length limits, but it is ideal to keep URLs under a few thousand characters.

[3] It's also possible to port compression libraries to JavaScript or WASM. But browsers have good implementations; we might as well use them!

[4] https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Compression

[5] https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Content-Encoding

[6] https://caniuse.com/mdn-api_compressionstream

[7] https://developer.mozilla.org/en-US/docs/Web/API/Compression_Streams_API