**The nuances of base64 encoding strings in JavaScript**



Matt Joseph

[GitHub](https://github.com/devadvance) [Homepage](https://mattj.io/)

base64 encoding and decoding is a common form of transforming binary content to be represented as web-safe text. It's used commonly for [data URLs](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URLs), such as inline images.

What happens when you apply base64 encoding and decoding to strings in JavaScript? This post explores the nuances and common pitfalls to avoid.

**btoa() and atob()**

The core functions to base64 encode and decode in JavaScript are [btoa()](https://developer.mozilla.org/docs/Web/API/btoa) and [atob()](https://developer.mozilla.org/docs/Web/API/atob). [btoa()](https://developer.mozilla.org/docs/Web/API/btoa) goes from a string to a base64-encoded string, and [atob()](https://developer.mozilla.org/docs/Web/API/atob) decodes back.

The following shows a quick example:

// A really plain string that is just code points below 128.  
const asciiString = 'hello';  
  
// This will work. It will print:  
// Encoded string: [aGVsbG8=]  
const asciiStringEncoded = btoa(asciiString);  
console.log(`Encoded string: [${asciiStringEncoded}]`);  
  
// This will work. It will print:  
// Decoded string: [hello]  
const asciiStringDecoded = atob(asciiStringEncoded);  
console.log(`Decoded string: [${asciiStringDecoded}]`);

Unfortunately, as [noted by the MDN docs](https://developer.mozilla.org/en-US/docs/Web/API/btoa#unicode_strings), this only works with strings that contain ASCII characters, or characters that can be represented by a single byte. In other words, this won't work with Unicode.

To see what happens, try the following code:

// Sample string that represents a combination of small, medium, and large code points.  
// This sample string is valid UTF-16.  
// 'hello' has code points that are each below 128.  
// '⛳' is a single 16-bit code units.  
// '❤️' is a two 16-bit code units, U+2764 and U+FE0F (a heart and a variant).  
// '🧀' is a 32-bit code point (U+1F9C0), which can also be represented as the surrogate pair of two 16-bit code units '\ud83e\uddc0'.  
const validUTF16String = 'hello⛳❤️🧀';  
  
// This will not work. It will print:  
// DOMException: Failed to execute 'btoa' on 'Window': The string to be encoded contains characters outside of the Latin1 range.  
try {  
  const validUTF16StringEncoded = btoa(validUTF16String);  
  console.log(`Encoded string: [${validUTF16StringEncoded}]`);  
} catch (error) {  
  console.log(error);  
}

Any one of the emojis in the string will cause an error. Why does Unicode cause this problem??

To understand, let's take a step back and understand strings, both in computer science and JavaScript.

**Strings in Unicode and JavaScript**

[Unicode](https://en.wikipedia.org/wiki/Unicode) is the current global standard for character encoding, or the practice of assigning a number to a specific character so that it can be used in computer systems. For a deeper dive into Unicode, visit [this W3C article](https://www.w3.org/International/talks/0505-unicode-intro/).

Some examples of characters in Unicode and their associated numbers:

* h - 104
* ñ - 241
* ❤ - 2764
* ❤️ - 2764 with a hidden modifier numbered 65039
* ⛳ - 9971
* 🧀 - 129472

The numbers representing each character are called "code points". You can think of "code points" as an address to each character. In the red heart emoji, there are actually two code points: one for a heart and one to "vary" the color and make it always red.

Learn more about the idea of [variant selectors](https://en.wikipedia.org/wiki/Variation_Selectors_(Unicode_block)).

Unicode has two common ways of taking these code points and making them into sequences of bytes that computers can consistently interpret: UTF-8 and UTF-16.

An oversimplified view is this:

* In UTF-8, a code point can use between one and four bytes (8 bits per byte).
* In UTF-16, a code point is always two bytes (16 bits).

Importantly, JavaScript processes strings as UTF-16. This breaks functions like btoa(), which effectively operate on the assumption that each character in the string maps to a single byte. This is stated explicitly on [MDN](https://developer.mozilla.org/en-US/docs/Web/API/btoa):

*The btoa() method creates a Base64-encoded ASCII string from a binary string (i.e., a string in which each character in the string is treated as a byte of binary data).*

Now you know that characters in JavaScript often require more than one byte, the next section demonstrates how to handle this case for base64 encoding and decoding.

**btoa() and atob() with Unicode**

As you now know, the error being thrown is due to our string containing characters that sit outside of a single byte in UTF-16.

Fortunately, the [MDN article on base64](https://developer.mozilla.org/docs/Glossary/Base64#the_unicode_problem) includes some helpful sample code for solving this "Unicode problem". You can modify this code to work with the preceding example:

// From https://developer.mozilla.org/en-US/docs/Glossary/Base64#the\_unicode\_problem.  
function base64ToBytes(base64) {  
  const binString = atob(base64);  
  return Uint8Array.from(binString, (m) => m.codePointAt(0));  
}  
  
// From https://developer.mozilla.org/en-US/docs/Glossary/Base64#the\_unicode\_problem.  
function bytesToBase64(bytes) {  
  const binString = String.fromCodePoint(...bytes);  
  return btoa(binString);  
}  
  
// Sample string that represents a combination of small, medium, and large code points.  
// This sample string is valid UTF-16.  
// 'hello' has code points that are each below 128.  
// '⛳' is a single 16-bit code units.  
// '❤️' is a two 16-bit code units, U+2764 and U+FE0F (a heart and a variant).  
// '🧀' is a 32-bit code point (U+1F9C0), which can also be represented as the surrogate pair of two 16-bit code units '\ud83e\uddc0'.  
const validUTF16String = 'hello⛳❤️🧀';  
  
// This will work. It will print:  
// Encoded string: [aGVsbG/im7PinaTvuI/wn6eA]  
const validUTF16StringEncoded = bytesToBase64(new TextEncoder().encode(validUTF16String));  
console.log(`Encoded string: [${validUTF16StringEncoded}]`);  
  
// This will work. It will print:  
// Decoded string: [hello⛳❤️🧀]  
const validUTF16StringDecoded = new TextDecoder().decode(base64ToBytes(validUTF16StringEncoded));  
console.log(`Decoded string: [${validUTF16StringDecoded}]`);

The following steps explain what this code does to encode the string:

1. Use the [TextEncoder interface](https://developer.mozilla.org/en-US/docs/Web/API/TextEncoder) to take the UTF-16 encoded JavaScript string and convert it to a stream of UTF-8-encoded bytes using [TextEncoder.encode()](https://developer.mozilla.org/docs/Web/API/TextEncoder/encode).
2. This returns a [Uint8Array](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Uint8Array), which is a less commonly used data type in JavaScript and is a subclass of the [TypedArray](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/TypedArray).
3. Take that Uint8Array and provide it to the bytesToBase64() function, which uses [String.fromCodePoint()](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/String/fromCodePoint) to treat each byte in the Uint8Array as a code point and create a string from it, which results in a string of code points that are can all be represented as a single byte.
4. Take that string and use [btoa()](https://developer.mozilla.org/docs/Web/API/btoa) to base64 encode it.

The decoding process is the same thing, but in reverse.

This works because the step between Uint8Array and a string guarantees that while the string in JavaScript is represented as a UTF-16, two-byte encoding, the code point that each two bytes represents is always less than 128.

This code works well under most circumstances, but it will silently fail in others.

**Silent failure case**

Use the same code, but with a different string:

// From https://developer.mozilla.org/en-US/docs/Glossary/Base64#the\_unicode\_problem.  
function base64ToBytes(base64) {  
  const binString = atob(base64);  
  return Uint8Array.from(binString, (m) => m.codePointAt(0));  
}  
  
// From https://developer.mozilla.org/en-US/docs/Glossary/Base64#the\_unicode\_problem.  
function bytesToBase64(bytes) {  
  const binString = String.fromCodePoint(...bytes);  
  return btoa(binString);  
}  
  
// Sample string that represents a combination of small, medium, and large code points.  
// This sample string is invalid UTF-16.  
// 'hello' has code points that are each below 128.  
// '⛳' is a single 16-bit code units.  
// '❤️' is a two 16-bit code units, U+2764 and U+FE0F (a heart and a variant).  
// '🧀' is a 32-bit code point (U+1F9C0), which can also be represented as the surrogate pair of two 16-bit code units '\ud83e\uddc0'.  
// '\uDE75' is code unit that is one half of a surrogate pair.  
const partiallyInvalidUTF16String = 'hello⛳❤️🧀\uDE75';  
  
// This will work. It will print:  
// Encoded string: [aGVsbG/im7PinaTvuI/wn6eA77+9]  
const partiallyInvalidUTF16StringEncoded = bytesToBase64(new TextEncoder().encode(partiallyInvalidUTF16String));  
console.log(`Encoded string: [${partiallyInvalidUTF16StringEncoded}]`);  
  
// This will work. It will print:  
// Decoded string: [hello⛳❤️🧀�]  
const partiallyInvalidUTF16StringDecoded = new TextDecoder().decode(base64ToBytes(partiallyInvalidUTF16StringEncoded));  
console.log(`Decoded string: [${partiallyInvalidUTF16StringDecoded}]`);

If you take that last character after decoding (�) and check its hex value, you'll find that it's \uFFFD rather than the original \uDE75. It doesn't fail or throw an error, but the input and output data has silently changed. Why?

**Strings vary by JavaScript API**

As described previously, JavaScript processes strings as UTF-16. But UTF-16 strings have a unique property.

Take the cheese emoji as an example. The emoji (🧀) has a Unicode code point of 129472. Unfortunately, the maximum value for a 16-bit number is 65535! So how does UTF-16 represent this much higher number?

UTF-16 has a concept called *surrogate pairs*. You can think of it this way:

* The first number in the pair specifies which "book" to search in. This is called a "[surrogate](https://en.wikipedia.org/wiki/UTF-16#U+D800_to_U+DFFF_(surrogates))".
* The second number in the pair is the entry in the "book".

As you might imagine, it could sometimes be problematic to only have the number representing the book, but not the actual entry in that book. In UTF-16, this is known as a *lone surrogate*.

This is particularly challenging in JavaScript, because some APIs work despite having lone surrogates while others fail.

In this case, you're using [TextDecoder](https://developer.mozilla.org/docs/Web/API/TextDecoder/TextDecoder) when decoding back from base64. In particular, the [defaults for TextDecoder](https://developer.mozilla.org/docs/Web/API/TextDecoder/TextDecoder#parameters) specify the following:

It defaults to **false**, which means that the decoder substitutes malformed data with a replacement character.

That � character you observed earlier, which is represented as \uFFFD in hex, is that replacement character. In UTF-16, strings with lone surrogates are considered "malformed" or "not well formed".

There are various web standards (examples [1](https://www.w3.org/TR/DOM-Level-3-Core/core.html#ID-C74D1578), [2](https://www.w3.org/TR/DOM-Level-3-Core/glossary.html#dt-16-bit-unit), [3](https://webidl.spec.whatwg.org/#idl-USVString), [4](https://infra.spec.whatwg.org/#scalar-value)) that exactly specify when a malformed string affects API behavior, but notably TextDecoder is one of those APIs. It is good practice to make sure that strings are well formed before doing text processing.

**Check for well-formed strings**

Very recent browsers now have a function for this purpose: [isWellFormed()](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/String/isWellFormed).

Browser Support

* 111
* 111
* 119
* 16.4

[Source](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/String/isWellFormed)

You can achieve a similar outcome by using [encodeURIComponent()](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent), which [throws an URIError error](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent#exceptions) if the string contains a lone surrogate.

The following function uses [isWellFormed()](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/String/isWellFormed) if it is available and [encodeURIComponent()](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent) if it is not. Similar code can be used to create a polyfill for isWellFormed().

// Quick polyfill since older browsers do not support isWellFormed().  
// encodeURIComponent() throws an error for lone surrogates, which is essentially the same.  
function isWellFormed(str) {  
  if (typeof(str.isWellFormed)!="undefined") {  
    // Use the newer isWellFormed() feature.  
    return str.isWellFormed();  
  } else {  
    // Use the older encodeURIComponent().  
    try {  
      encodeURIComponent(str);  
      return true;  
    } catch (error) {  
      return false;  
    }  
  }  
}

**Put it all together**

Now that you know how to handle both Unicode and lone surrogates, you can put everything together to create code that handles all cases and does so without silent text replacement.

// From https://developer.mozilla.org/en-US/docs/Glossary/Base64#the\_unicode\_problem.  
function base64ToBytes(base64) {  
  const binString = atob(base64);  
  return Uint8Array.from(binString, (m) => m.codePointAt(0));  
}  
  
// From https://developer.mozilla.org/en-US/docs/Glossary/Base64#the\_unicode\_problem.  
function bytesToBase64(bytes) {  
  const binString = String.fromCodePoint(...bytes);  
  return btoa(binString);  
}  
  
// Quick polyfill since Firefox and Opera do not yet support isWellFormed().  
// encodeURIComponent() throws an error for lone surrogates, which is essentially the same.  
function isWellFormed(str) {  
  if (typeof(str.isWellFormed)!="undefined") {  
    // Use the newer isWellFormed() feature.  
    return str.isWellFormed();  
  } else {  
    // Use the older encodeURIComponent().  
    try {  
      encodeURIComponent(str);  
      return true;  
    } catch (error) {  
      return false;  
    }  
  }  
}  
  
const validUTF16String = 'hello⛳❤️🧀';  
const partiallyInvalidUTF16String = 'hello⛳❤️🧀\uDE75';  
  
if (isWellFormed(validUTF16String)) {  
  // This will work. It will print:  
  // Encoded string: [aGVsbG/im7PinaTvuI/wn6eA]  
  const validUTF16StringEncoded = bytesToBase64(new TextEncoder().encode(validUTF16String));  
  console.log(`Encoded string: [${validUTF16StringEncoded}]`);  
  
  // This will work. It will print:  
  // Decoded string: [hello⛳❤️🧀]  
  const validUTF16StringDecoded = new TextDecoder().decode(base64ToBytes(validUTF16StringEncoded));  
  console.log(`Decoded string: [${validUTF16StringDecoded}]`);  
} else {  
  // Not reached in this example.  
}  
  
if (isWellFormed(partiallyInvalidUTF16String)) {  
  // Not reached in this example.  
} else {  
  // This is not a well-formed string, so we handle that case.  
  console.log(`Cannot process a string with lone surrogates: [${partiallyInvalidUTF16String}]`);  
}

There are many optimizations that can be made to this code, such as generalizing into a polyfill, changing the TextDecoder parameters to throw instead of silently replacing lone surrogates, and more.

With this knowledge and code, you can also make explicit decisions about how to handle malformed strings, such as rejecting the data or explicitly enabling data replacement, or perhaps throwing an error for later analysis.

In addition to being a valuable example for base64 encoding and decoding, this post provides an example as to why careful text processing is particularly important, especially when the text data is coming from user-generated or external sources.

Was this helpful?