UTF-8 in JavaScript

Working with string encodings in JavaScript can sometimes be frustrating. My latest frustrations came when working with the `atob` and `btoa` browser functions. These functions convert between a binary string and a Base64 encoded ASCII string. But they blow up when faced with Unicode:

```
> btoa('\u0227');
Error: INVALID_CHARACTER_ERR: DOM Exception 5
```

The MDN docs on `btoa` point to the following functions from Johan Sundström for working with Unicode:

```
function encode_utf8( s ) {
  return unescape( encodeURIComponent( s ) );
}

function decode_utf8( s ) {
  return decodeURIComponent( escape( s ) );
}
```

Armed with these functions, `btoa` gives the expected result:

```
> btoa(encode_utf8('\u0227'));
"yKc="
```

Johan's functions work, but they feel like a bit of black magic. So I decided to delve into each part of the `encode_utf8` function to understand why it works.

encodeURIComponent

The `encode_utf8` function starts with

`encodeURIComponent`. `encodeURIComponent` is defined in the latest ECMAScript spec:

> The encodeURIComponent function computes a new version of a URI in which each instance of certain characters is replaced by one, two, three, or four escape sequences representing the UTF-8 encoding of the character.

What struck me right off the bat is that `encodeURIComponent` is tied to UTF-8. Its not like similar functions other languages where the encoding can be specified as a function argument. This is fine if you want to work with UTF-8, but not if you want to work with other encodings (although there are ways to manage fixed-length encodings using ArrayBuffers).

Here's an example of using `encodeURIComponent` on the Unicode character U+0227 (ȧ, LATIN SMALL LETTER A WITH DOT ABOVE):

```
> encodeURIComponent('\u0227');
"%C8%A7"
```

The result is a percent-encoded string of each byte of the UTF-8 representation of the character. Looking at the documentation for Unicode character U+0227, we can verify that the UTF-8 representation in hex is indeed 0xC8 0xA7.

The ECMAScript definition is vague; it states that `encodeURIComponent` replaces "certain characters", without defining what those characters are. But a quick hack shows that any character over 0x7E is encoded, so I think it's safe to say that any non-ASCII character will be encoded by `encodeURIComponent`. Since percent-encoding only uses the numbers 0-9, letters A-F and the '%' character, the resulting string is guaranteed to be ASCII.

Now we could stop here. All `btoa` needs is an ASCII string, which `encodeURIComponent` provides:

```
> btoa(encodeURIComponent('\u0227'));
"JUM4JUE3"
```

But there are a few drawbacks to this. The first is that the resulting string is not the same bytes as the input string. It is an encoded representation of the original bytes. This could make things annoying or difficult when interoperating with other systems.

The second drawback is that `encodeURIComponent` produces a larger string. A single Unicode character could take up to 4 bytes in UTF-8, which would produce a URI-encoded string of 12 characters. After Base64 encoding (which makes strings larger), the output string would be much larger than the input. In order to tame the string size, Johan turns to the `unescape` function.

unescape

Unlike `encodeURIComponent`, the `escape`/`unescape` functions are NOT defined by the latest version of ECMAScript. According to JavaScript: The Definitive Guide (6th Edition, page 855):

> Although unescape was standardized in the first verison of ECMAScript, it has been deprecated and removed from the standard by ECMAScript v3. Implementations of ECMAScript are likely to implement this function, but they are not required to.

But seeing as how these functions have been a part of browsers from almost the beginning, I doubt they will disappear anytime soon.

The `unescape` function unescapes a percent-encoded string. It works with the standard percent-encoding (%XX) as well as the non-standard Unicode (%uXXXX) percent encoding. Here's the result of unescaping the string from the original example:

```
> var e = encodeURIComponent('\u0227');
console.log(e);
```

```
"%C8%A7"
> var u = unescape(e); console.log(u);
"È§"
```

"È§" are the ASCII characters for C8 and A7, respectively:

```
> u.charCodeAt(0).toString(16);
"c8"
> u.charCodeAt(1).toString(16);
"a7"
```

Calling `unescape` is very different from calling `decodeURIComponent`. `decodeURIComponent` interprets the string with UTF-8, which means it would combine the two encoded characters back into their original UTF-8 representation. `unescape` merely returns the characters without any interpretation.

The benefits of using `unescape` is that we now have a representation of the actual UTF-8 bytes, and it is a smaller string:

```
> e.length
6
> u.length
2
```

The bytes in the new string are the ASCII-representation of the bytes in the UTF-8 representation of the original string.

Conclusion

Putting it all together, here are the transformations the string goes through in `encode_utf8`:

Original Unicode string	encodeURIComponent	unescape
à	%C8%A7	È§

Johan's bit of black magic works because it turns a

Unicode string into an encoded UTF-8 string, and then returns the ASCII representation of the encoded bytes. What I love about this method is that it uses the fact that browser's have multiple encoding functions to its advantage. Using only the pair of `encodeURIComponent/decodeURIComponent` or `escape/unescape` would be a no-op; the functions would cancel each other out and return the original string. The genius of Johan's method is that it uses the two different encoding functions in tandem to get what we need.

Posted on 20 Jul 2012 by Monsur Hossain