ecmanaut

Webby thoughts, most about around interesting applications of ecmascript in relation to other open web standards. I live in Mountain View, California, and spend some of my spare time co-maintaining Greasemonkey together with Anthony Lieuallen.

2006-07-10

Encoding / decoding UTF8 in javascript

From time to time it has somewhat annoyed me that UTF8 (today's most common Unicode transport encoding, recommended by the IETF) conversion is not readily available in browser javascript. It really *is*, though, I realized today:

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

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

2012 Update: Monsur Hossain took a moment to explain how and why this works. It's a good, in-depth post citing all standards in play so you need not bring a wizard's beard to know why it works. Executive summary: `escape` and `unescape` operate solely on octets, encoding/decoding %XXs only, while `encodeURIComponent` and `decodeURIComponent` encode/decode to/from UTF-8, and *in addition* encode/decode %XXs. The hack above combines both tools to cancel out all but the UTF-8 encoding/decoding parts, which happen inside the heavily optimized browser native code, instead of you pulling the weight in javascript.

Tested and working like a charm in these browsers:

Win32

- Firefox 1.5.0.6
- Firefox 1.5.0.4
- Internet Explorer 6.0.2900.2180
- Opera 9.0.8502

MacOS

- Camino 2006061318 (1.0.2)
- Firefox 1.5.0.4
- Safari 2.0.4 (419.3)

Any modern standards compliant browser should handle this code, though, so don't worry that it's a rather sparse test matrix. But feel free to use my test case: encoding and decoding the word "räksmörgås". That's incidentally Swedish for a shrimp sandwich, by the way -- very good subject matter indeed.

And if you hand me your platform/browser combination and the its success/failure status for the tests, I'll try to update the post accordingly.
Categories:

- javascript
- unicode
- utf8
- tips

Posted by Johan Sundström at 15:47

ELSEWHERE

BLOG ARCHIVE

- ► 2013 (3)
- ► 2012 (5)
- ► 2011 (14)
- ► 2010 (17)
- ► 2009 (11)
- ► 2008 (15)
- ► 2007 (24)
- ▼ 2006 (72)
  - ► December (3)
  - ► November (3)
  - ► October (3)
  - ► September (2)
  - ► August (4)
  - ▼ July (6)
    - A peek at Yahoo! UI
    - Google × Dilbert mashup
    - Spam fighting phpBB boards
    - Custom keymaps in windows
    - Encoding / decoding UTF8 in javascript
    - Expressive user scripts with XPath and E4X
  - ► June (3)
  - ► May (7)
  - ► April (3)
  - ► March (8)
  - ► February (8)
  - ► January (22)
- ► 2005 (87)
- ► 2004 (2)

+5 Recommend this on Google

43 comments

★ 14

Leave a message...

Best Community My Disqus 4 Share

Johan Sundström • 5 years ago
In that case you are doing it wrong;
unescape(encodeURIComponent("€")) === "\xE2\x82\xAC" and **decodeURIComponent(escape("\xE2\x82\xAC")) === "€"** both return true, as they they are supposed to.
7 • Reply • Share ›

Simon • 4 years ago
Still now in 2010, this is the only usable search result on solving this specific problem. Have nobody else noticed and written about this?

Also an UPDATE: Works in Chrome --
Win32, Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US) AppleWebKit/532.5 (KHTML, like Gecko) Chrome/4.1.249.1045 Safari/532.5
4 • Reply • Share ›

Mathias Bynens • 5 months ago
Just a heads up: this approach doesn’t work for unmatched surrogate halves, since the `URI` functions throw errors on those. It works fine for all Unicode symbols except high surrogates (code points 0xD800 to 0xDBFF) and low surrogates (code points 0xDC00 to 0xDFFF).
3 • Reply • Share ›

Johan Sundström Mod → Mathias Bynens • 5 months ago
While true, for most purposes that is more of a feature than an issue. Surrogate pairs being how ecmascript (well, UTF-16) maps Unicode chars beyond the basic multilingual plane (code points > 0xFFFF) to double-codepoint pairs (making, say, "0123456789".length be 20, not 10), it could even be argued that not throwing an error while encoding or decoding half of such a character would be more problematic.

Still, it's a good thing to be aware of. I seem to recall Chrome or some other modern browser was working on a native real charset codec functionality as a language or browser object model feature, but haven't been able to backtrack to read up on it. I'd be interested in their take, if I didn't just dream it up. (And link to it from here as a topically relevant alternative.)
• Reply • Share ›

Alan Hogan • 2 years ago
If anyone wants to see a bit of proof that this actually is doing exactly the right thing, and producing a string of bytes equivalent to a UTF-8 binary string… here’s this jsFiddle: http://jsfiddle.net/alanhogan/...
3 • Reply • Share ›

Technics · 5 years ago
Hi ,I believe that this method doesn't work for characters like the EURO symbol €. In Firefox I get Malformed URI sequence error
3 ^ | ˅ · Reply · Share ›

Johan Sundström · 5 years ago
Same principles as in all programming apply: don't decode UTF-8 that isn't; don't divide by zero, and so on. Just adding a try/catch will hide errors in input and is inadvisable.
2 ^ | ˅ · Reply · Share ›

Pekka Klärck · 23 days ago
We recommended this approach for JSXCompressor project [1] when we noticed their UTF-8 decode method did not work correctly with characters outside the Unicode BMP [2]. Unfortunately this solution turned out to be rather slow with bigger strings and especially IE8 became totally unusable.

Luckily we found a nice UTF-8 decode algorithm implemented with C [3] and were able to translate it to Javascript. This algorithm is now used in the latest versions of JXG.Util.UTF8. You may want to consider using it if performance is important and using an external library is not an issue.

[1] http://jsxgraph.uni-bayreuth.d...
[2] https://github.com/jsxgraph/js...
[3] http://bjoern.hoehrmann.de/utf...
1 ^ | ˅ · Reply · Share ›

Paul · 3 years ago
Thanks!

We had been sending UTF-8 data through jQuery which was dealing with it fine in all browsers, until we switched to running our app in a Webkit component embedded in a PyQt application. Still sending UTF-8 from Python but it goes through an implicit "eval()" call, and I was ending up with Â£ (capital A, circumflex accent, pound sterling symbol) instead of £ (pound sterling symbol).

Popping the UTF-8 strings through the above has fixed this.

This is the QtWebkit 4.6.2.0

Win32, Mozilla/5.0 (Windows; U; Windows NT 5.1; en-GB) AppleWebKit/532.4 (KHTML, like Gecko) Qt/4.6.2 Safari/532.4

Thanks again
1 ^ | ˅ · Reply · Share ›

Thomas Frank · 7 years ago
Nifty!
1 ^ | ˅ · Reply · Share ›

David Zentgraf · 2 years ago
This really needs some clarification on what it actually does. One cannot simply "encode" or "decode" UTF-8. One can only *convert* from one encoding to another encoding. "Decode UTF-8" to what? "Encode UTF-8" from what?

It's a shame that this nonsense page has such a high ranking on Google for the keywords " Javascript UTF-8". BTW, all strings in

Google for the keywords "Javascript UTF-8". BTW, all strings in Javascript are UTF-8 by design, period.

1 ^ | 1 ⌄ • Reply • Share ›

Johan Sundström Mod → David Zentgraf • 2 years ago

Deconfusion time.

These functions decode to and from the javascript's native character set, which is UCS-2, which can represent the basic multilingual plane of Unicode (i e the subset of Unicode code points that fit in 16 bits of storage without special coding).

You can think of a javascript string as "an array of 16-bit integers with a length property", representing anything you want (examples: text, html, xml, images, encoded in any of a zillion encodings, including the UTF family of text encodings, gzip compression, whatever) but if you pass one to alert or document.write, their contents are treated as text from the Unicode basic multilingual plane, or Unicode code points, all in the range \u0000 to \uFFFF inclusive. If you run encode_utf8 on that, the output is a range of such characters, transport encoded as UTF-8 (in other words, each output octet is in the range \u0001 to \u00FF, and conform to the UTF-8 encoding). My decode_utf8 function above has its domain and range reversed, decoding UTF-8 strings to non-transport-encoded javascript native strings.

All the UTF-* family of encodings are *transport encodings* of (all of) Unicode. UTF-* are particularly unsuited as internal string representations if you ever care for random access of string contents, or telling string length in number of characters or code points, as you can't index the n:th character of an UTF-8 encoded text in O(1) time, since each code point is represented as one or more bytes (dependent on its Unicode code point) or words in memory, which would make an operation such as "öh".length == 2 an O(n) operation, as "ö" internally would be represented (in UTF-8) as a two-byte sequence and "h" a single byte. Fortunately, no javascript engine I am aware of uses it, and as far as I know, they all use UCS-2, which encodes each character as its 16-bit Unicode code point (and can't represent any of the code points outside of the basic multilingual plane; the full Unicode character set currently uses 21-bit code points). With UCS-2, measuring string length in number of characters is simply dividing the in-memory representation's length in octets by two, without needing to scan its contents with a counter incremented at the completion of each code point. (This does not apply for any of the UTF-* encodings.)

What internal method of string storage a javascript implementation uses is irrelevant as long as it fulfills the language specification, as long as you can't access and expose that internal representation in any fashion, so making claims like "all strings in javascript are UTF-8 by design" just adds confusion. Even if they were, you wouldn't be able to benefit from it by passing them to entities expecting UTF-8 input, unless we had a string.rawUTF8 property or similar that exposed its (internally) encoded representation.

The point of these two functions thus is to produce the encoded and decoded versions of the UTF-8 transport encoding, given input that is raw Unicode and UTF-8 encoded Unicode respectively.

7 ^ | ⌄ · Reply · Share ›

Oliver Mannion · 9 months ago

escape and unescape are now deprecated functions - what would you recommend using instead to do ut8 encoding/decoding? custom javascript?

^ | ⌄ · Reply · Share ›

Johan Sundström Mod → Oliver Mannion · 9 months ago

I think you misunderstand what "deprecated" means (and implies) in the context of the web. Ever since the introduction of encodeURIComponent and decodeURIComponent, escape and unescape have effectively been deprecated for the purposes they were initially intended for: escaping strings for use in URLs, which they were never very good for, as they only worked for a subset of the characters javascript strings can employ.

Browser vendors can't cut backwards compatibility with them without breaking code out there that uses them, and probably won't, since the implementation is trivial and small anyway.

A few years to a decade from now, when presently emerging standards for javascript charset conversion have matured and are available in all major browsers you can switch to that, but for the forseeable future, I recommend sticking with these.

And if you really worry about browsers cutting back compat, you can provide your own fallbacks like this before you start using the utf8 encoder/decoder:

```
if (typeof escape !== 'function') window.escape = functic
  function q(c) {
    c = c.charCodeAt();
    return '%' + (c<16 ? '0' : '') + c.toString(16).toUpp
  }
  return s.replace(/[\x00-),:-?[-^`{-\xFF]/g, q);
};
if (typeof unescape !== 'function') window.unescape = fur
  function d(x, n) {
    return String.fromCharCode(parseInt(n, 16));
  }
  return s.replace(/%([0-9A-F]{2})/i, d);
};
```

1 ^ | ⌄ · Reply · Share ›

Marius Dumitru Bughiu · 2 years ago

This removes new lines \r\n
Any fix for this?

^ | ⌄ · Reply · Share ›

Johan Sundström Mod → Marius Dumitru Bughiu · 2 years ago

It doesn't; the context in which you're using it is. Maybe you're copy/pasting from a form field that doesn't preserve newlines.

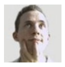
To convince yourself, look at http://jsfiddle.net/V6yGL/1/ which doesn't touch the \r\n:s.

• Reply • Share ›

Daan • 2 years ago

How would I send an URL like http://someurl.com/send?variab... ? The unescape function would always change the %26 to an emphasis which cause the ' stuff' part to be cut off, not?

• Reply • Share ›



Johan Sundström Mod → Daan • 2 years ago

No. Your imagined problem would happen if the code read "return unescape(s)". The beauty of the hack is that encodeURIComponent(s) converts the % character to %25, which unescape then converts back to %, leaving it unchanged, while for all code points in the input string that is represented differently in UTF-8, encodeURIComponent spits out two or more %XX sequences, which unescape proceeds to decode to the same number of raw octets. decode_utf8 does the same, but backwards.

• Reply • Share ›

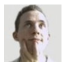
mikr00 • 3 years ago

Thanks very much for this very cool solution. I have one Problem though: I use your code on a website with an input-text-field whose value is read via javascript and than handed over to a php site via post. Your solution works perferct with german umlauts (äöü) but it does not work with the "ß". Do you have any idea, why this could be the case.

Thanks a lot!

Michael

• Reply • Share ›



Johan Sundström Mod → mikr00 • 3 years ago

It's probably a cut and paste issue, as encode_utf8('ß') === '\xC3\x9F', which looks like 'Ã', because unicode (and iso-8859-1, which is the subset of code points 00...FF) code point 9F is non-printable (not defined, actually), so some input sanitization somewhere might strip it for you in the process.

If you try to paste javascript:prompt(1,unescape(encodeURIComponent('ß'))) into your address field and immediately use ctrl-c (or cmd-c) to copy it without making any selection changes, you may get the non-printable too (works for me here when testing; if I paste it into Firebug and execute 'Ã'.length, it reports the proper 2 as expected, not 1, as it would if you failed to copy the non-printable).

The same problem comes up a lot if you are using asian characters like Chinese, Japanese or Korean glyphs; my räksmörgås example was kind of benevolent in the sense that it's copy-paste friendly. People often miss that UTF-8 is a binary encoding; it just often happens to look like it's always going to output "readable" text. These functions are supposed to be talking with code, not with humans. :-)

1 • Reply • Share ›

Simon S • 3 years ago

I can't get it working. I have login form on a UTF-8 page and need to convert the password to ISO-8859-1 before posting to a ISO-8859-1 server. The conversion gets done, but it doesn't seem to be real ISO-8859-1, as the login fails because of wrong converted special characters (I tried it with the german umlaut "ä").

• Reply • Share ›

Johan Sundström Mod → Simon S • 3 years ago

I think you are looking for the wrong kind of solution. You'd probably rather want to furnish your form with an accept-charset attribute, most likely with a value of "ISO-8859-1".

• Reply • Share ›

Your Name • 3 years ago

Thanks for the handy hack. It may look tricky but is actually ensured by standards. I'm throwing away my hand rolled UTF-8 encoder now.

• Reply • Share ›

shdanfo • 3 years ago

Thanks for your note on decodeURIComponent. I'm dealing with an XSLT template that includes something like following text

• Reply • Share ›

donjohn • 4 years ago

Very good! Thank you very much. I've been having problems with a facebook connect site where the facebook stream.publish api method was not recognizing the accented characters being sent through a javascript function. I couldn't find any suggestions anywhere until I came across this blog. I applied your suggestion and voilà! problem resolved!

• Reply • Share ›

bucabay • 4 years ago

Best solution I've seen so far. The others usually are bitwise operations to get the UTF-8 byte sequences, but don't fully implement the encoding.

• Reply • Share ›

Johan Sundström • 4 years ago

Well, somewhere between your UTF-8 encoders and this blog, something is going wrong at least, because „ and – and … (code points 8222, 8211 and 8230 respectively, all way beyond 255) are not 8-bit characters, which every octet in a valid UTF-8 string must be, by definition.

Maybe you are sitting on some Windows system doing fancy quotes under your feet, or similar muck. Best of luck with your debugging.

• Reply • Share ›

Conny • 4 years ago

Hmm, well... If RÄKSMÖRGÅS is encoded to utf8 with Javascript, it gets RÃ„KSMÃ–RGÃ…S. But if the same word is encoded to utf8 with Java or UltraEdit Text Editor (ASCII to UTF-8), it gets RÃ„KSMÃ–RGÃ…S.

I am parsing an XML-document encoded as utf8 (the Java-version...) in Javascript.

So, is there two versions of utf8?

• Reply • Share ›

Johan Sundström • 4 years ago

That is because the UTF-8 encoding of RÄKSMÖRGÅS is RÃKSMÃRGÃS, not RÃ„KSMÃ–RGÃ…S. (Any incorrect encoded input sequence will probably give you that error or a similar one.)

• Reply • Share ›

Conny • 4 years ago

How about uppercase letters?

For me your example rÃ¤ksmÃ¶rgÃ¥s is correctly decoded into räksmörgås but RÃ„KSMÃ–RGÃ…S gives a "malformed URI sequence" error.

• Reply • Share ›

Anonymous • 5 years ago

Result of **your platform/browser**:
Win32, Mozilla/5.0 (Windows; U; Windows NT 5.1; de; rv:1.9.0.8) Gecko/2009032609 Firefox/3.0.8 (.NET CLR 3.5.30729)

My $_SERVER['HTTP_USER_AGENT']:
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1) ; .NET CLR 2.0.50727; .NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)

Remarks:
I am running internet explorer version 8.0, which is operating in some sort of compatability mode, and suddenly the executed javascript calls ajax-requests as xmlHttp=new XMLHttpRequest() when before it (iex 6 and 7)) would execute xmlHttp=new ActiveXObject("Msxml2.XMLHTTP") or xmlHttp=new ActiveXObject("Microsoft.XMLHTTP")

take care!

• Reply • Share ›

Anonymous • 5 years ago

internet explorer 8:
Win32, Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1) ; .NET CLR 2.0.50727; .NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)

• Reply • Share ›

scott • 5 years ago

Excellent stuff. I wanted to note that:
decode_utf8()
can throw an error. It is probably a good idea to wrap the call in a try...catch

I think I was getting this error when trying to decode UTF-8 when it was really ISO 8859.

• Reply • Share ›

Johan Sundström • 6 years ago
Put it between a **<script>** and **</script>** tag and you'll be fine.

• Reply • Share ›

Anonymous • 6 years ago
Just one question... does it matter where you put that code? I'm completely new to javascript, so i don't have any idea.

• Reply • Share ›

Anonymous • 6 years ago
Hi Johan

Thanks for the great little UTF-8 hack :) I've used on my open source tool Hackvertor:-
http://www.businessinfo.co.uk/...

btw this isn't comment spam, Johan gave me permission to plug my tool :)

• Reply • Share ›

Riši • 6 years ago
Got it, thanks.

• Reply • Share ›

Johan Sundström • 6 years ago
What did you expect? That is how the UTF8 encoded text is represented when the undecoded UTF8 message is seen as a normal eight-bit string, when your character set is ISO-8859-1, commonly referred to as Latin-1.

• Reply • Share ›

Riši • 6 years ago
Thanks Johan.
However, on my machine, this does not quite work: the word got encoded as
rÃ¤ksmÃ¶rgÃ¥s

Win32, Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.8.1.8) Gecko/20071008 Firefox/2.0.0.8

Any idea why?

• Reply • Share ›

Thomas Langvann • 7 years ago
Worked like a charm with:
Linux Firefox/2.0.0.2 (Ubuntu-edgy)

• Reply • Share ›

Andrej • 7 years ago
Best solution, thanks.

• Reply • Share ›

Johan Sundström • 7 years ago
Your are in luck! Transforming text in ISO 8859-1 to Unicode is the

identity transform (as in no change at all), as the code points they share have the same meaning in both encodings. For all other encodings (save US ASCII, in part a subset ISO 8859-1), you need to resort to laborious replace() hacks.

• Reply • Share ›

Alex Iskold • 7 years ago

How about the case when the original text is encoded using ISO-8859-1?

Thanks!

Alex

• Reply • Share ›

Subscribe Add Disqus to your site

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Powered by Blogger.