Why is there trailing garbage when I try to decode the bytes of a HttpContent object?

devblogs.microsoft.com/oldnewthing/20210923-00

September 23, 2021

Raymond Chen

A customer was having trouble extracting text from an HTTP response.

```
winrt::HttpRequest request = ...;

auto result = co_await request.Content().ReadAsStringAsync();
```

This version produced a string that looked mostly okay, but some parts were corrupted.

> `{"name":"ðAPCDCS±meow"}`

From inspection, it's clear that what we have here is mojibake, wherein a UTF-8 string is being misinterpreted in some other 8-bit character set.

According to RFC 2616 section 3.7.1, if there is no explicit character set for a `text` media subtype, the default character set is ISO-8859-1. Evidently, this server returned a string encoded as UTF-8 but failed to indicate this character set when it reported its `Content-Type`. As a result, the string defaults to ISO-8859-1.

Oops.

Now, section 3.4.1 of RFC 2616 acknowledges that it is common for HTTP clients to interpret the lack of an explicit character set as an opportunity to make a best guess. The Windows Runtime does perform some guessing of the character set if no character set is provided:

- If the buffer begins with a UTF-8 BOM, a UTF-16LE BOM, a UTF-16BE BOM, or a GB18030 BOM, then the buffer is decoded according to that character set.
- If the content type is `application/json` or is of the form `*+json`, then it is decoded as UTF-8.
- Otherwise, it is decoded as ISO-8859-1.

The fact that we made it all the way to the last step means that the server didn't use a UTF-8 BOM, nor did it set the content type to `application/json` even though it was returning JSON.

Oops.

Okay, so let's try to work around this (apparently very broken) server by taking the response bytes and explicitly decoding them as UTF-8.

```
std::wstring Utf8ToUtf16(char const* str)
{
    std::wstring result;
    if (str) {
        auto resultLen = MultiByteToWideChar(
            CP_UTF8, MB_ERR_INVALID_CHARS, str, -1, nullptr, 0);
        if (resultLen) {
            result.resize(resultLen);
            MultiByteToWideChar(
                CP_UTF8, MB_ERR_INVALID_CHARS, str, -1,
                result.data(), resultLen);
        }
    }
    return result;
}

winrt::HttpRequest request = ...;

auto buffer = co_await request.Content().ReadAsBufferAsync();
auto result = Utf8ToUtf16((char const*)buffer.data());
```

This version worked better, but it had garbage at the end:

> `{"name":"meow"}SOH`

In this case, the problem was not in the acquisition of the buffer but rather in the conversion of the buffer to a string. The `buffer.data()` method returns a pointer to the start of the buffer, and the code passes this as the source string to `MultiByteToWideChar` with `-1` as the string length.

The special value `-1` means that the pointer should be treated as the start of a null-terminated string. But the `Buffer` that is produced by `ReadAsBufferAsync` is just raw bytes returned from the server, and the server isn't going to put a null terminator at the end. The server says, "The response is 19 bytes long," and it sends the 19 bytes and that's that.

So the extra garbage is a read buffer overflow, where the code just reads past the end of the buffer until it finally runs into a zero byte somewhere.

You want to decode the bytes in the buffer, so you need to specify the number of bytes in the buffer, rather than saying "Just keep decoding until you hit a zero byte."

```
std::wstring Utf8ToUtf16(char const* str, int32_t inputLen)
{
    std::wstring result;
    if (str) {
        auto resultLen = MultiByteToWideChar(
            CP_UTF8, MB_ERR_INVALID_CHARS, str, inputLen, nullptr, 0);
        if (resultLen) {
            result.resize(resultLen);
            MultiByteToWideChar(
                CP_UTF8, MB_ERR_INVALID_CHARS, str, inputLen,
                result.data(), resultLen);
        }
    }
    return result;
}

winrt::HttpRequest request = ...;

auto buffer = co_await request.Content().ReadAsBufferAsync();
auto result = Utf8ToUtf16(
    (char const*)buffer.data(),
    static_cast<int32_t>(buffer.Length()));
```

This produces the desired string, decoded as UTF-8, with no garbage.

Now, it turns out that you don't have to write the code to take a `Buffer` containing a string encoded in UTF-8 and convert it to a UTF-16 string. The Windows Runtime already provides a helper function to do this:

```
winrt::HttpRequest request = ...;

auto buffer = co_await request.Content().ReadAsBufferAsync();
auto result = CryptographicBuffer::ConvertBinaryToString(
    BinaryStringEncoding::Utf8, buffer);
```

But since we're using C++/WinRT, we can avoid all that and use the conversion built into C++/WinRT we learned last time. The hard part is getting a `std::string_view` out of a `buffer`.

```
winrt::HttpRequest request = ...;

auto buffer = co_await request.Content().ReadAsBufferAsync();
auto result = winrt::to_hstring(
    std::string_view{
        static_cast<char const*>(buffer.data()),
        buffer.Length() });
```

So there you go, reading the raw buffer and converting it from UTF-8 to a UTF-16 string.

In the meantime, go fix your server already.

Epilogue: The customer found that indeed their server was misconfigured. The response was being generated via a loopback server, and it was putting the `Content-Type` header in the `ResponseHeaders` instead of the `ContentHeaders`.

Raymond Chen

Follow