Property-based tests of Buffer Character Encodings

Student:
Rasmus Byskov Gluud – raglu16@student.sdu.dk

Aims

The aim of this project is to test the properties of the Character Encoding operators in the Buffer Class of Node.js.

Introduction

Pure JavaScript is Unicode friendly, but it does not handle binary data very well. This is fine on a browser, but when dealing with TCP streams or reading or writing to system files, it is sometimes necessary to handle streams of binary data. One approach to manipulating, creating, or consuming binary data in JavaScript is to use *Buffers*.

A Buffer is a class in Node.js that is designed to handle raw binary data. A Buffer is like an array of data in raw memory that are allocated outside V8 heap. Because of this, the data in these arrays are stored as bytes expressed in hexadecimal notation and cannot be resized. Buffers offer different ways of encoding JavaScript strings into binary data. These encodings are UTF-8, UTF-16LE, ASCII, Latin1, Base64, and Hexadecimal.[1]

The buffer class is global and can be initialized with the following statement:

```
> var buffer = Buffer.from('Some String','ascii');
```

The above line allocates space for the array and populates it with binary encoding of the string in the first argument as specified in the second argument. Logging the buffer to the console prints out the raw hexadecimal values of the ASCII-string:

```
> console.log(buffer)
<Buffer 53 6f 6d 65 20 53 74 72 69 6e 67>
```

Testing generator

This project uses Fast-check and Jest as testing framworks.

Fast-check

Fast-check is a property-based testing framework for JavaScript. Fast-check behaves much like QuickCheck in that it can perform random testing along with shrinking failure cases.[2]

Jest

Jest is a JavaScript testing framework that can integrate along with Fast-check. The only reason this testing framework was integrated into this project is due to its familiarity and simplicity.[3]

---

[1] https://nodejs.org/en/knowledge/advanced/buffers/how-to-use-buffers/
[2] https://github.com/dubzzz/fast-check
[3] https://github.com/facebook/jest

Experiments, Results, and Discussion

Traditionally, a report is divided into sections that separates the experimental setups, their results, and what to make of it. In this report however, each experiment is its own section and includes the setup, results, as well as discussion within. I found this format to be the most easily readable for this document.

The experiments in this project are inspired by the various property-based testing approaches presented by Scott Wlaschin.[4] These approaches focus on the domain-specific 'logical' properties, which include checking properties such as combining operations in different orders, using inverse operations, verifying idempotence, and applying structural induction.

A common theme for each experiment is that the tests are proceeded inside one or more for-loops since they will do the same operations, but with different encodings. However, special cases for hexadecimal and Base64 encodings were used. This is because a Buffer with hexadecimal encoding expresses its values as literal hexadecimal characters, which excludes characters outside the range from '*0-9*' and '*a-f*'; furthermore, each character must form a pair in the string otherwise the last character will be discarded. Base64 encoded strings must have padding added if its length is not a multiple of four. If properties like these are not handled the testing suites will falsely consider them failures. Fortunately, Fast-check can generate strings that only contain hexadecimal characters and strings that include Base64 padding.

One of the main challenges in this project were eliminating false positives such as the aforementioned issue, which unfortunately also complicates the readability of the test files.

Idempotency

What happens if a string is encoded twice? Idempotence is a property in which an operation can be applied multiple times without changing the results of the initial application. Ideally a string encoded twice should produce the same set of data. This experiment encodes the appropriate string and then encodes it again while checking if they both match. The result for this experiment is shown below:

```
C:\Users\rasmu\Desktop\buffer-encoding-tests\tests>jest idempotence.test.js
 PASS  ./idempotence.test.js
  √ utf8 (32 ms)
  √ utf16le (9 ms)
  √ latin1 (11 ms)
  √ ascii (12 ms)
  √ hex (30 ms)
  √ base64 (9 ms)

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        1.315 s
Ran all test suites matching /idempotence.test.js/i.
```

Structural induction

What happens if a string is encoded in multiple substrings? This experiment checks to see if an encoded string matches the sum of its encoded substrings. Hexadecimal and Base64 were again considered here, since they cannot be divided into smaller charArrays than a length of 2 or 4. The experiment did also not produce any errors:

[4] https://fsharpforfunandprofit.com/posts/property-based-testing-2/

```
C:\Users\rasmu\Desktop\buffer-encoding-tests\tests>jest structural_induction.test.js
 PASS  ./structural_induction.test.js
  √ utf8 (21 ms)
  √ utf16le (13 ms)
  √ latin1 (12 ms)
  √ ascii (10 ms)
  √ hex (30 ms)
  √ base64 (10 ms)

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        1.484 s
Ran all test suites matching /structural_induction.test.js/i.
```

Inverse operation

What happens if a string encoding is converted back to its initial encoding? The purpose of this experiment is to check if an encoded string will be the same if it is converted into another encoding and back again. The result of this experiment proved at least 4 failure cases:

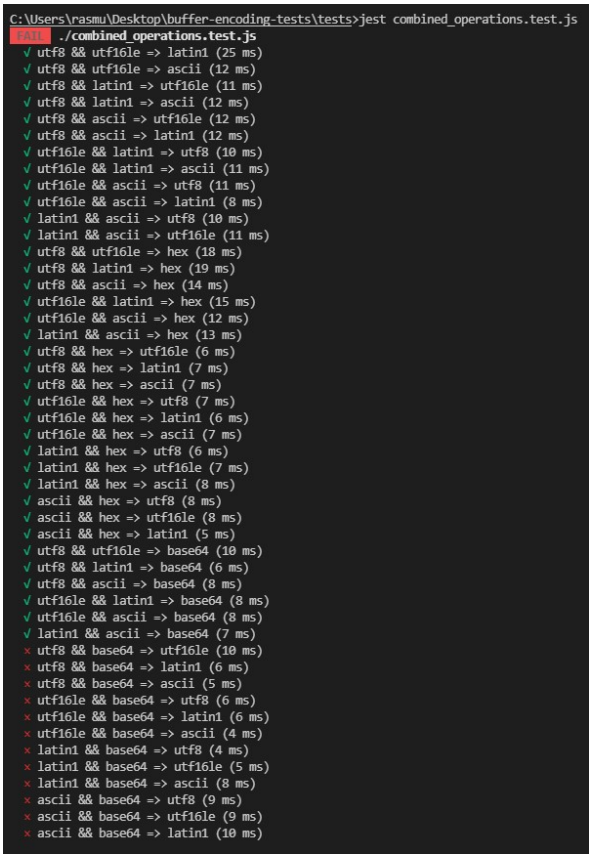
```
C:\Users\rasmu\Desktop\buffer-encoding-tests\tests>jest inverse_operation.test.js
 FAIL  ./inverse_operation.test.js
  √ utf8 => utf16le => utf8 (23 ms)
  √ utf8 => latin1 => utf8 (11 ms)
  √ utf8 => ascii => utf8 (11 ms)
  √ utf16le => utf8 => utf16le (11 ms)
  √ utf16le => latin1 => utf16le (10 ms)
  √ utf16le => ascii => utf16le (10 ms)
  √ latin1 => utf8 => latin1 (8 ms)
  √ latin1 => utf16le => latin1 (12 ms)
  √ latin1 => ascii => latin1 (10 ms)
  √ ascii => utf8 => ascii (10 ms)
  √ ascii => utf16le => ascii (9 ms)
  √ ascii => latin1 => ascii (11 ms)
  √ utf8 => hex => utf8 (15 ms)
  √ utf16le => hex => utf16le (15 ms)
  √ latin1 => hex => latin1 (16 ms)
  √ ascii => hex => ascii (13 ms)
  √ hex => utf8 => hex (11 ms)
  √ hex => utf16le => hex (14 ms)
  √ hex => latin1 => hex (13 ms)
  √ hex => ascii => hex (12 ms)
  × utf8 => base64 => utf8 (9 ms)
  × utf16le => base64 => utf16le (5 ms)
  × latin1 => base64 => latin1 (4 ms)
  × ascii => base64 => ascii (6 ms)
  √ base64 => utf8 => base64 (6 ms)
  √ base64 => utf16le => base64 (11 ms)
  √ base64 => latin1 => base64 (9 ms)
  √ base64 => ascii => base64 (6 ms)
```

```
Test Suites: 1 failed, 1 total
Tests:       4 failed, 24 passed, 28 total
Snapshots:   0 total
Time:        1.633 s
Ran all test suites matching /inverse_operation.test.js/i.
```

In this test the first counterexample was the initial UTF-8 string value “AX==” which changed into “Aw==” after converting into Base64 and back again. The following counterexamples in this test were similar in nature: “AB==”, “AR==”, and “AH==”. The reason behind phenomenon this is unknown, but it smells buggy.

Combined operations

What happens if two different encodings of the same string are converted to a third encoding? This experiment checks to see if two encodings of the same string can produce the exact same string in a different encoding. The experiment shows similar results as the Inverse Operations; any conversion from Base64 produced the same errors:



The errors in the Base64 encoding can be reduced to a single digestible line that can be run from the console:

```
> Buffer.from('AB==','base64').toString('base64')
'AA=='
```

Attempts were made for figuring out how this supposed error occurs in the Node.js source code, but the root cause was not found.

Conclusion

The character encoding conversion in the Buffer class of Node.js has through property-based testing been proven exceed its stated operation. However, it has one easily reproducible error when converting a string into Base64 encoding.