JavaScript Code Snippets

Release 0.1.0

Shailesh Kumar

Apr 09, 2022

Contents:

1 Language Features **1**
- 1.1 Closures . . . 1
- 1.2 Async Await . . . 1

2 Text Processing **3**
- 2.1 String Validations . . . 3

3 Working with Operating System in Node **5**
- 3.1 Buffers . . . 5
- 3.2 File System . . . 6

4 Working with REST APIs **7**

5 Amazon Web Services **9**

6 Image Processing **11**

7 Working with Email **13**

8 Express Web Framework **15**
- 8.1 Hello World . . . 15
- 8.2 Working with Requests . . . 16
- 8.3 Creating Responses . . . 16

9 Functional Programming **17**
- 9.1 Empty . . . 17
- 9.2 Search . . . 18

10 Real Time Applications with Socket IO **19**

11 Miscellaneous Topics **21**
- 11.1 Date and Time with Moment JS . . . 21
- 11.2 Cron Jobs . . . 21
- 11.3 Random Numbers . . . 21

12 Indices and tables **23**

CHAPTER 1

Language Features

1.1 Closures

1.2 Async Await

CHAPTER 2

Text Processing

2.1 String Validations

CHAPTER 3

Working with Operating System in Node

3.1 Buffers

Pure JavaScript doesn't have support for binary data. NodeJS has a separate data type called `Buffer` which provides support for binary data (an array of bytes).

This is useful for working with file system and TCP/IP networking.

Creating a buffer

A buffer of 5 bytes without initialization:

```
let data = new Buffer(5);
```

Creating a buffer from an array of values (values should be between 0 to 255):

```
let data = new Buffer([10 10 20 20 50]);
```

Buffers and Strings

Converting a sting into a buffer:

```
let data = new Buffer('hello text');
```

By default, strings are encoded in buffers using `utf-8` encoding.

Converting a string into a buffer using a specific encoding:

```
let data = new Buffer('hello text', 'utf16le');
data = new Buffer('hello text', 'utf-8');
data = new Buffer('hello text', 'ascii');
data = new Buffer('hello text', 'base64');
```

3.2 File System

NodeJS comes with an extensive file system API in a module called `fs`. However, the module is traditionally based on callbacks. In this book, we will be focused on writing codes using promise based APIs which can be easily used in `async/await` paradigm. Hence, we will use a library named fs-extra.

Checking existence of a file

```
const fs = require('fs-extra');

(async () => {
    let filename = './fs.rst';
    let exists = await fs.pathExists(filename);
    if (exists){
        console.log(`The file ${filename} exists.`);
    } else {
        console.log(`The file ${filename} does not exist.`);
    }
})();
```

Reading a file

```
const fs = require('fs-extra');

(async () => {
    // The readFile function reads the contents of a file in a buffer.
    let contents = await fs.readFile('read_file.js');
    // Buffer can be converted into string for further processing.
    let contents_str = contents.toString();
    console.log(contents_str);
})();
```

CHAPTER 4

Working with REST APIs

CHAPTER 5

Amazon Web Services

CHAPTER 6

Image Processing

CHAPTER 7

Working with Email

CHAPTER 8

Express Web Framework

8.1 Hello World

Make sure that you have installed the express framework:

```
npm install --save express
```

We require the express package:

```
const express = require('express');
```

We create an express application instance:

```
const app = express();
```

A web application essentially handles incoming requests at different (URL) endpoints and returns data (in form of HTML/text/JSON etc.) as response. To achieve this, we define different request handlers. A request handler is a function with two arguments (`request` and `response`). The request object captures all information about incoming HTTP request. The response object provides methods for sending response HTTP headers and data.

Here is a simple request handler function:

```
const index = function (request, response) {
    response.send('Hello World!');
}
```

We now tell the express framework about the endpoint for which this request handler will be used:

```
app.get('/', index);
```

We choose port number for the express web application:

```
const port = 3000;
```

We now setup the express application to listen at the specified port. If the listening starts successfully, then a callback will get called informing us about it:

```
app.listen(port, function() {
    console.log(`Example app listening on port ${port}!`);
});
```

Here is the complete code. Please save it in a file named `helloworld.js`.

```
const express = require('express');
const app = express();
const port = 3000;

const index = function (request, response) {
    response.send('Hello World!');
}

app.get('/', index);

app.listen(port, function() {
    console.log(`Example app listening on port ${port}!`);
});
```

We can start it by running:

```
node helloworld.js
```

Time to head to http://localhost:3000 and see the result.

8.2 Working with Requests

8.3 Creating Responses

CHAPTER 9

Functional Programming

Major libraries supporting functional programming in JavaScript

- Ramda
- Lodash

We will be primarily using Ramda for our examples below.

Importing the library:

```
R = require('ramda');
```

9.1 Empty

Checking whether something is empty (an object or an array or a string):

```
> R.isEmpty({})
true
> R.isEmpty([])
true
> R.isEmpty('')
true
```

This function will appropriately return `false` in other cases:

```
> R.isEmpty(0)
false
> R.isEmpty(1)
false
> R.isEmpty(true)
false
> R.isEmpty(false)
false
```

(continues on next page)

(continued from previous page)

```
> R.isEmpty(null)
false
> R.isEmpty(undefined)
false
> R.isEmpty(NaN)
false
> R.isEmpty({1: 2})
false
> R.isEmpty([1])
false
> R.isEmpty('a')
false
```

9.2 Search

Searching for the index of an object in a array:

```
> R.findIndex(x => x == 2, [1, 2, 3])
1
```

The index is 0 based. The first argument is a predicate which returns true when a suitable search criterion is satisfied. Here, we are looking for the first element in the array whose value is 2. Hence the criterion is `x == 2`. The predicate function is written as an arrow function `x => x ==2`.

When no element of the array satisfies the given predicate, it returns -1:

```
> R.findIndex(x => x == 2, [1, 4, 3])
-1
```

The index of the first match is returned always:

```
> R.findIndex(x => x == 2, [1, 2, 2, 3])
1
```

Searching in an array of objects based on the value of an attribute:

```
> R.findIndex(x => x.v == 2, [{v : 4}, {v : 3}, {v: 2}])
2
```

Finding the array element:

```
> R.find(x => x == 2, [1, 2, 3])
2
> R.find(x => x == 2, [1, 4, 3])
undefined
> R.find(x => x == 2, [1, 2, 2, 3])
2
> R.find(x => x.v == 2, [{v : 4}, {v : 3}, {v: 2}])
{ v: 2 }
```

If there is no array element satisfying the criterion in the predicate, then `undefined` is returned.

CHAPTER 10

Real Time Applications with Socket IO

CHAPTER 11

Miscellaneous Topics

11.1 Date and Time with Moment JS

11.2 Cron Jobs

11.3 Random Numbers

CHAPTER 12

Indices and tables

- genindex
- modindex
- search