CSCI 4140 – Tutorial 8

Learning the basics of Node.js

Matt YIU, Man Tung (mtyiu@cse)

SHB 118

Office Hour: Wednesday, 3-5 pm

2016.03.17

Outline

- What is Node.js?

- Learning the basics of Node.js: Non-blocking I/O, HTTP

  – Exercises adapted from `learnyounode`:
     https://github.com/rvagg/learnyounode

License of `learnyounode`

learnyounode is Copyright (c) 2013-2015 learnyounode contributors (listed above) and licenced under the MIT licence. All rights not explicitly granted in the MIT license are reserved. See the included LICENSE.md file for more details.

learnyounode builds on the excellent work by @substack and @maxogden who created stream-adventure which serves as the original foundation for **learnyounode**.

What is Node.js?

- An **open-source**, **cross-platform** runtime environment for **server-side** and **networking applications**

- Applications are written in **JavaScript**
  - Node.js uses **Google V8 JavaScript engine** to execute code

- Provide an **event-driven architecture** and a **non-blocking I/O** API
  - One process for all **concurrent connections**
  - Optimizes an application's **throughput** and **scalability**
  - For your information, Apache uses **process-/thread-based architecture**, which is relatively inefficient
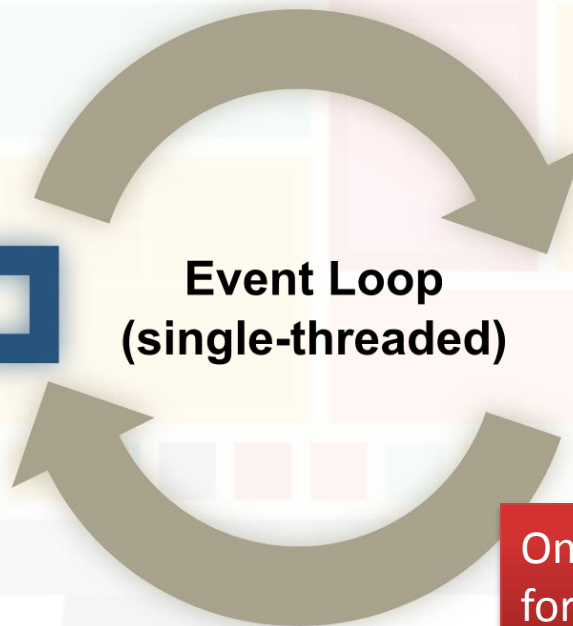   - A new process / thread is created per connection

What is Node.js: Event-driven architecture

For those who have taken CSCI/CENG 3150…

Event Emitters

Event Queue

Event Loop (single-threaded)
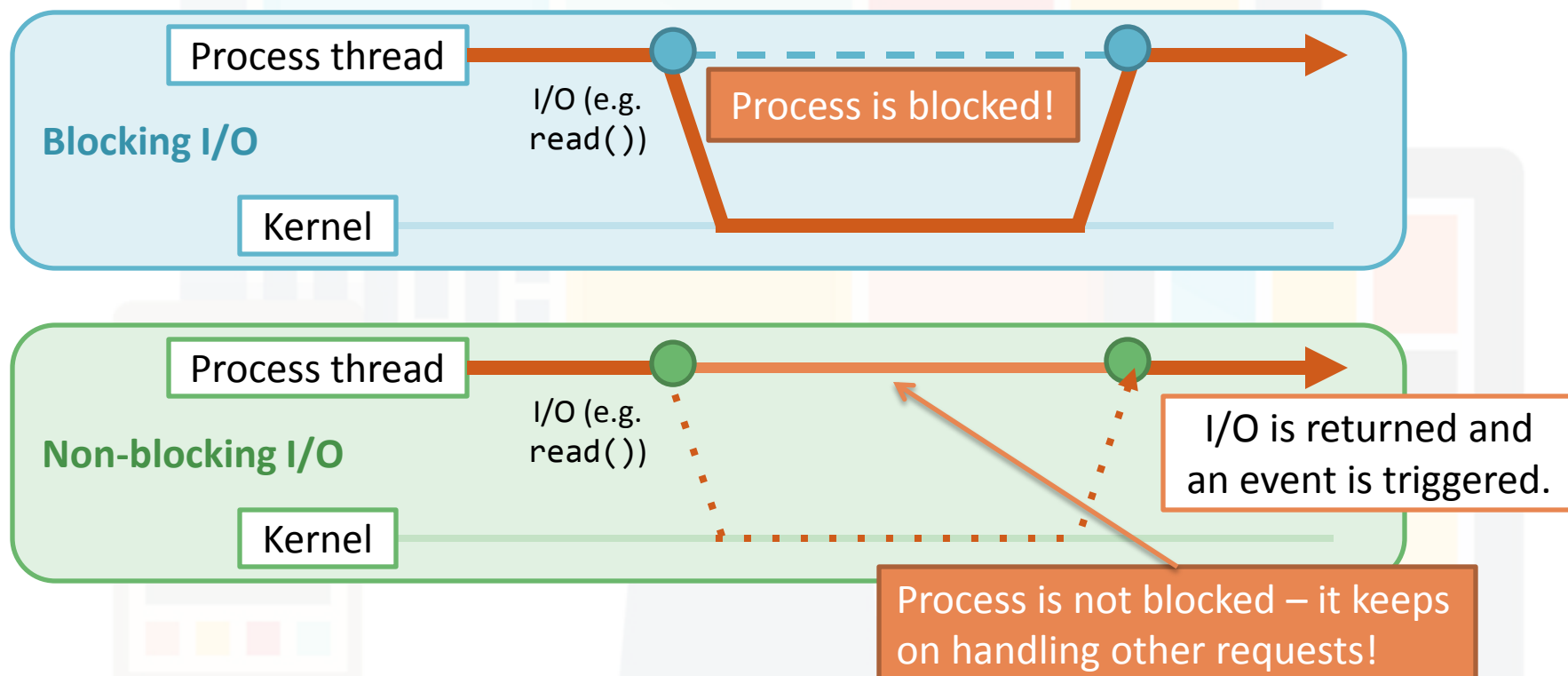
States

Event Handler

One thread is enough for all connections!

Reference: http://berb.github.io/diploma-thesis/original/042_serverarch.html

What is Node.js: Non-blocking I/O

- Also called **Asynchronous I/O**
- You are familiar with **blocking I/O** already…

For those who have taken CSCI/CENG 3150…

Process thread

Blocking I/O

I/O (e.g. read())

Process is blocked!

Kernel

Process thread

Non-blocking I/O

I/O (e.g. read())

I/O is returned and an event is triggered.

Kernel

Process is not blocked – it keeps on handling other requests!

Node.js HTTP server

- HTTP is a **first class citizen** in Node
  - Forget about Apache / IIS / Nginx
- Say "Hello World!" with Node.js HTTP server:
  - Execute "**node nodejs/server.js**" in your terminal and visit **http://127.0.0.1:4140/** in your browser

```
var http = require( 'http' );
http.createServer( function( request, response ) {
    response.writeHead( 200, { 'Content-Type' : 'text/plain' } );
    response.end( 'Hello World!\n' );
} ).listen( 4140, '127.0.0.1' );

console.log( 'Server running at http://127.0.0.1:4140/' );
```

nodejs/server.js

= **Important**

Learning the basics of Node.js: Non-blocking I/O, HTTP

Exercises adapted from https://github.com/rvagg/learnyounode

Exercise 1: Hello World

- *Let's learn Node.js by doing exercises!*

- **Problem:** Write a program that prints the text "HELLO WORLD" to the console (`stdout`)

- Use the **console** API: http://nodejs.org/api/console.html

```
console.log( "HELLO WORLD" );
```
nodejs/ex1-hello.js

```
$ node nodejs/ex1-hello.js
```
Terminal

- Useful for debugging
  - Obviously you cannot call "`alert()`"…

Exercise 2: Baby steps

- **Problem:** Write a program that accepts one or more numbers as command-line arguments and prints the sum of those numbers to the console (`stdout`)

- Access **command-line arguments** from the **argv** property of the global **process** object

  – For example, executing "**node program.js 1 2 3**"

program.js	`console.log(process.argv);`

Output	`[ 'node', '/home/mtyiu/program.js', '1', '2', '3' ]`

- Note that the command-line arguments are strings

  – Convert the string into number with "**Number(<string>)**"

Solution

Exercise 2: Baby steps

```
var sum = 0;
for( var i = 2; i < process.argv.length; i++ )
    sum += Number( process.argv[ i ] );
console.log( sum );
```

nodejs/ex2-baby-steps.js

Solution

Exercise 2: Baby steps

process.argv is an array so you can get its length in the field "length".

```
var sum = 0;
for( var i = 2; i < process.argv.length; i++ )
    sum += Number( process.argv[ i ] );
console.log( sum );
```

nodejs/ex2-baby-steps.js

Parse the command-line arguments into numbers.

Exercise 3: My first I/O

- **Problem:** Write a program that uses a single **synchronous** filesystem operation to read a file and print the **number of newlines** it contains to the console (`stdout`), similar to running **`cat file | wc -l`**.

- We need the **`fs`** module from the Node core library
  - http://nodejs.org/api/fs.html
  - Load the `fs` module into a **variable**: `var fs = require( 'fs' );`

- All synchronous (or blocking) filesystem methods end with "Sync", e.g., "**fs.readFileSync(** *<file path>* **)**"
  - This method returns a **Buffer** object containing the complete contents of the file

Exercise 3: My first I/O

- **Buffer** objects are Node's way of efficiently representing arbitrary arrays of data
  - To convert them to strings, call "**toString()**" method on them, e.g., **var str = buf.toString()**
- To count the number of newlines in a string, you can split it using the "**split()**" method with the "**\n**" character as the delimiter
- Remember that the last line of the input file does not contain a newline

Exercise 3: My first I/O

Solution

```
var fs = require( 'fs' );
var buf = fs.readFileSync( process.argv[ 2 ] );
var str = buf.toString();
console.log( str.split( '\n' ).length - 1 );
```

nodejs/ex3-first-io.js

Exercise 3: My first I/O

Solution

Use the synchronous filesystem API to read the file.

```
var fs = require( 'fs' );
var buf = fs.readFileSync( process.argv[ 2 ] );
var str = buf.toString();
console.log( str.split( '\n' ).length - 1 );
```

nodejs/ex3-first-io.js

The last line does not contain a newline so it is not counted.

Exercise 4: My first asynchronous I/O

- **Problem:** Write a program that uses a single **asynchronous** filesystem operation to read a file and print the number of **newlines** it contains to the console (`stdout`), similar to running `cat file | wc -l`.

- `fs.readFile()` is the asynchronous version of `fs.readFileSync()`
  - This method returns **without blocking**
  - To read the file contents, you need to pass a **callback function** which will be called when the I/O completes
    - This concept is **extremely important** in **JavaScript** programming!

Exercise 4: My first asynchronous I/O

- The callback function should have the following signature:

```
function callback ( err, data ) { /* ... */ }
```

Represent an error

The Buffer object / string containing the file contents

- **fs.readFile()** function accepts two or three arguments:

```
fs.readFile( filename[, options], callback )
```

Pass "utf8" for the options argument to get a string instead of an Buffer object

Exercise 4: My first asynchronous I/O

Solution

```
var fs = require( 'fs' );
fs.readFile(
    process.argv[ 2 ],
    'utf8',
    function( err, data ) {
        console.log( data.split( '\n' ).length - 1 );
    }
);
```

nodejs/ex4-first-async-io.js

Exercise 4: My first asynchronous I/O

Solution

```
var fs = require( 'fs' );
fs.readFile(
    process.argv[ 2 ],
    'utf8',
    function( err, data ) {
        console.log( data.split( '\n' ).length - 1 );
    }
);
```

nodejs/ex4-first-async-io.js

This function is only executed after the I/O operation completes.
The `readFile()` call will not be blocked.

Exercise 5: Filtered `ls`

- **Problem:** Create a program that prints a list of files in a given directory to the console using **asynchronous I/O**, filtered by the **extension** of the files
  - *1st argument*: A directory name
  - *2nd argument*: A file extension to filter by
- Similar to Exercise 4, but with **`fs.readdir()`**
  - http://nodejs.org/api/fs.html#fs_fs_readdir_path_callback
- You will also need **`path.extname()`** in the **`path`** module
  - http://nodejs.org/api/path.html#path_path_extname_p

Exercise 5: Filtered `ls`

Solution

```
var fs = require( 'fs' );
var path = require( 'path' );
var ext = '.' + process.argv[ 3 ];

fs.readdir( process.argv[ 2 ], function( err, data ) {
    data.forEach( function( i ) {
        if ( path.extname( i ) == ext )
            console.log( i );
    } );
} );
```

nodejs/ex5-filtered-ls.js

Exercise 5: Filtered `ls`

Solution

```
var fs = require( 'fs' );
var path = require( 'path' );
var ext = '.' + process.argv[ 3 ];

fs.readdir( process.argv[ 2 ], function( err, data ) {
    data.forEach( function( i ) {
        if ( path.extname( i ) == ext )
            console.log( i );
    } );
} );
```

nodejs/ex5-filtered-ls.js

For `fs.readdir()`, `data` is an array of the names of the files in the directory excluding '`.`' and '`..`'.

Instead of the ordinary for-loop which iterates from 0 to `data.length`, we use **`data.forEach()`** to iterate over all values in the array.
You need to provide a callback function which takes three optional arguments: (1) Current value, (2) Index & (3) The array `forEach()` was called upon.

Exercise 6: Make it modular

- **Problem:** Same as Exercise 5, but you need to make it **modular**

- Write a **module file** to do most of the work
  - The module should **export a function** which takes 3 arguments:
    1. The directory name
    2. The filename extension string (identical to the corresponding command-line argument)
    3. A callback function
  - The callback function should use the idiomatic node `(err, data)` convention
    - **err** is `null` if there is no errors; return the errors from `fs.readdir()` otherwise
    - **data** is the filtered list of files, as an Array
  - Nothing should be printed from your module file
    - Only print from the original program

Exercise 6: Make it modular

- From the problem statement, we induce the four requirements of a module:

  – **Export** a single function that takes exactly the **arguments** described

  – Call the **callback** exactly **once** with an error or some data as described

  – **Don't change anything** else, like global variables or `stdout`

  – Handle all the **errors** that may occur and pass them to the callback

    - Do **early-returns** within callback functions if there is an error

- A good Node.js developer should follow these rules!

Exercise 6: Make it modular

- In the module file (e.g., `module.js`), assign a function to the **`module.exports`** object to define a **single function export**:

```
module.exports = function (args) { /* ... */ }
```

- In your program, load the module (`module.js`) using the **`require()`** call ("`./`" indicates that it is a local module):

```
var module = require( './module' );
```

  – **Note:** "`.js`" can be omitted

- The `require()` call returns what you export in the module file
  – In this example, it returns a function that you can call directly!

Exercise 6: Make it modular

Solution

```
var fs = require( 'fs' );
var path = require( 'path' );

module.exports = function( dirname, ext, callback ) {
    var ext = '.' + ext;
    fs.readdir( dirname, function( err, data ) {
        if ( err )
            return callback( err );

        var ret = data.filter( function( i ) {
            return ( path.extname( i ) == ext );
        } );
        callback( null, ret );
    } );
};
```

nodejs/ex6-make-it-modular-module.js *(Module file)*

Exercise 6: Make it modular

Solution

```
var fs = require( 'fs' );
var path = require( 'path' );

module.exports = function( dirname, ext, callback ) {
    var ext = '.' + ext;
    fs.readdir( dirname, function( err, data ) {
        if ( err )
            return callback( err );

        var ret = data.filter( function( i ) {
            return ( path.extname( i ) == ext );
        } );
        callback( null, ret );
    } );
};
```

Early-return

The `filter()` method creates a new array with all elements that pass the test implemented by the provided function.

nodejs/ex6-make-it-modular-module.js *(Module file)*

Exercise 6: Make it modular

Solution

```
var module = require( './ex6-make-it-modular-module' );
module( process.argv[ 2 ], process.argv[ 3 ], function( err, data ) {
    if ( err )
        console.error( "Error: " + err );

    data.forEach( function( i ) {
        console.log( i );
    } );
} );
```

nodejs/ex6-make-it-modular.js *(Main program)*

Exercise 6: Make it modular

Solution

Load a local module

```
var module = require( './ex6-make-it-modular-module' );
module( process.argv[ 2 ], process.argv[ 3 ], function( err, data ) {
    if ( err )
        console.error( "Error: " + err );

    data.forEach( function( i ) {
        console.log( i );
    } );
} );
```

Report errors

nodejs/ex6-make-it-modular.js *(Main program)*

Exercise 7: HTTP client

- **Problem:** Write a program that performs an **HTTP GET request** to a URL provided to you as the first command-line argument. Write the `String` contents of each "`data`" event from the response to a new line on the console (`stdout`).
  - **Note:** There is a sample scenario in Assignment 2 – retrieving video title from YouTube server using an HTTP GET request

- Use the **`http.get()`** method in the **`http`** module
  - http://nodejs.org/api/http.html#http_http_get_options_callback
  - *1st argument*: The URL you want to GET
  - *2nd argument*: A callback with the following signature:

    ```
    function callback ( response ) { /* ... */ }
    ```

Exercise 7: HTTP client

- The **response** object is a **Node Stream** object
  - It is an object that emits events
  - Register an **event listener** (`.on(*, callback)`) to handle the event
    - This is the core of "**event-driven architecture**"
  - For `http.get()`, the three events that are of most interests are: "**data**", "**error**" and "**end**"
    - See http://nodejs.org/api/http.html#http_http_incomingmessage and http://nodejs.org/api/stream.html#stream_class_stream_readable

- The `response` object has a **setEncoding()** method
  - If you call this method with "`utf8`", the **data** events emit `Strings` instead of the standard Node **Buffer** objects

Exercise 7: HTTP client

Solution

```
var http = require( 'http' );
http.get( process.argv[ 2 ], function( response ) {
    response.setEncoding( 'utf8' );
    response
        .on( 'data', console.log )
        .on( 'error', console.error );
} );
```

nodejs/ex7-http-client.js

Exercise 7: HTTP client

Solution

```
var http = require( 'http' );
http.get( process.argv[ 2 ], function( response ) {
    response.setEncoding( 'utf8' );
    response
        .on( 'data', console.log )
        .on( 'error', console.error );
} );
```

nodejs/ex7-http-client.js

console.log() and console.error() are functions! You don't need to wrap it in your own functions as the event listener.

The ".on(<event name>, <listener/callback>)" method returns the emitter (e.g., response in this example). Therefore this call can be chained.

Exercise 8: HTTP collect

- **Problem:** Write a program that performs an HTTP GET request to a URL provided to you as the first command-line argument. Collect **all data** from the server (not just the first "`data`" event) and then write two lines to the console (`stdout`).
  - *1st line*: The **number of characters** received from the server
  - *2nd line*: The **complete `String` of characters** sent by the server

- Two approaches:
  - Collect and append data across multiple "`data`" events. Write the output when an "end" event is emitted
  - Use a **third-party package** to abstract the difficulties involved in collecting an entire stream of data, e.g., `bl` and `concat-stream`

Exercise 8: HTTP collect

- Let's try the second approach to explore an important component in Node.js: **npm** – the **package manager** for node
  - FYI, the package manager for Python is `pip`

- To install the Node package `bl`, type in the terminal:

  > `$ npm install bl`

  - npm will download and install the latest version of the package into a subdirectory named `node_modules`

- When you write "`var bl = require('bl');`" in your program, Node will first look in the **core modules**, and then in the `node_modules` directory where the package is located.

- Read https://www.npmjs.com/package/bl for its usage

Exercise 8: HTTP collect

Solution

```javascript
var http = require( 'http' );
var bl = require( 'bl' );
http.get( process.argv[ 2 ], function( response ) {
    response.pipe(
        bl(
            function( err, data ) {
                if ( err )
                    return console.error( "Error: " + err );
                data = data.toString();
                console.log( data.length );
                console.log( data );
            }
        )
    );
} );
```

nodejs/ex8-http-collect.js

Note that `data` is a Buffer object so you need to convert it to `String` with `toString()`.

Exercise 9: Juggling async

- **Problem:** Same as Exercise 8, but this time you will be provided with **3 URLs** as the first 3 command-line arguments
  - Print the complete content provided by each of the URLs to the console (`stdout`), one line per URL
  - No need to print out the length
  - The content must be printed out in the **same order** as the URLs are provided to you as command-line arguments

- **This exercise is tricky!**
  - `http.get()` is an asynchronous call
  - The callback function is executed when any of the servers response
  - The responses will probably be **out of order**!
    - You need to **queue the results** and print the data when all data is ready

Exercise 9: Juggling async

Solution

```
var http = require( 'http' );
var bl = require( 'bl' );
```

Modifying from the solution of Exercise 8...

```
    http.get( process.argv[ 2 ], function( response ) {
        response.pipe( bl( function( err, data ) {
            if ( err )
                return console.error( "Error: " + err );
            data = data.toString();
            console.log( data.length );
            console.log( data );



        } ) );
    } );
```

nodejs/ex9-juggling-async-wrong.js

Exercise 9: Juggling async

Solution

```javascript
var http = require( 'http' );
var bl = require( 'bl' );
var ret = [];
var count = 0;
for ( var i = 2; i < 5; i++ ) {
    http.get( process.argv[ i ], function( response ) {
        response.pipe( bl( function( err, data ) {
            if ( err )
                return console.error( "Error: " + err );
            ret[ i - 2 ] = data.toString();
            count++;



        } ) );
    } );
}
```

Modifying from the solution of Exercise 8...

Iterate over the 3 URLs and save the responses to the ret array. Count the number of responses received in count.

nodejs/ex9-juggling-async-wrong.js

Exercise 9: Juggling async

Solution

```
var http = require( 'http' );
var bl = require( 'bl' );
var ret = [];
var count = 0;
for ( var i = 2; i < 5; i++ ) {
    http.get( process.argv[ i ], function( response ) {
        response.pipe( bl( function( err, data ) {
            if ( err )
                return console.error( "Error: " + err );
            ret[ i - 2 ] = data.toString();
            count++;

            if ( count == 3 ) {
                for ( var j = 0; j < 3; j++ )
                    console.log( ret[ j ] );
            }
        } ) );
    } );
}
```

Modifying from the solution of Exercise 8…

Print the data to the console when all 3 responses are received.

nodejs/ex9-juggling-async-wrong.js

Exercise 9: Juggling async

Wrong Solution

```javascript
var http = require( 'http' );
var bl = require( 'bl' );
var ret = [];
var count = 0;
for ( var i = 2; i < 5; i++ ) {
    http.get( process.argv[ i ], function( response ) {
        response.pipe( bl( function( err, data ) {
            if ( err )
                return console.error( "Error: " + err );
            ret[ i - 2 ] = data.toString();
            count++;

            if ( count == 3 ) {
                for ( var j = 0; j < 3; j++ )
                    console.log( ret[ j ] );
            }
        } ) );
    } );
}
```

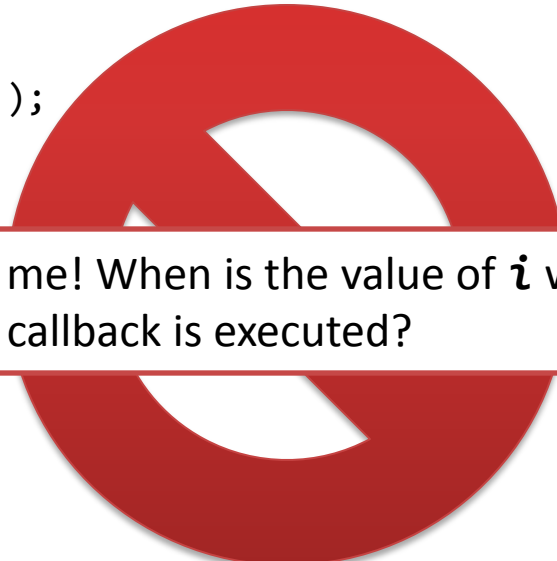Sorry...this solution is wrong! Why?

nodejs/ex9-juggling-async-wrong.js

Exercise 9: Juggling async

Wrong Solution

```
var http = require( 'http' );
var bl = require( 'bl' );
var ret = [];
var count = 0;
for ( var i = 2; i < 5; i++ ) {
    http.get( process.argv[ i ], function( response ) {
        response.pipe( bl( function( err, data ) {
            if ( err )
                return console.error( "Error: " + err );
            ret[ i - 2 ] = data.toString();
            count++;

            if ( count == 3 ) {
                for ( var j = 0; j < 3; j++ )
                    console.log( ret[ j ] );
            }
        } ) );
    } );
}
```

Tell me! When is the value of i when the callback is executed?

nodejs/ex9-juggling-async-wrong.js

Exercise 9: Juggling async

<div style="float:right">**Wrong Solution**</div>

```javascript
var http = require( 'http' );
var bl = require( 'bl' );
var ret = [];
var count = 0;
for ( var i = 2; i < 5; i++ ) {
    http.get( process.argv[ i ], function( response ) {
        response.pipe( bl( function( err, data ) {
            if ( err )
                return console.error( "Error: " + err );
            ret[ i - 2 ] = data.toString();
            count++;

            if ( count == 3 ) {
                for ( var j = 0; j < 3; j++ )
                    console.log( ret[ j ] );
            }
        } ) );
    } );
}
```

The value of i is **evaluated** only **when the callback is executed**. Since the for-loop has probably completed, i **= 5 no matter which callback is executed**.

nodejs/ex9-juggling-async-wrong.js

Exercise 9: Juggling async

Right Solution

Modifying from the wrong solution…

```
var http = require( 'http' );
var bl = require( 'bl' );
var ret = [];
var count = 0;
for ( var i = 2; i < 5; i++ ) {
    http.get( process.argv[ i ],

            function( response ) {
          response.pipe( bl( function( err, data ) {
            if ( err )
                return console.error( "Error: " + err );
            ret[    i - 2 ] = data.toString();
            count++;

            if ( count == 3 ) {
                for ( var j = 0; j < 3; j++ )
                    console.log( ret[ j ] );
            }
        } ) );
      }
      );
}
```

nodejs/ex9-juggling-async-right.js

Exercise 9: Juggling async

Right Solution

Modifying from the wrong solution…

```
var http = require( 'http' );
var bl = require( 'bl' );
var ret = [];
var count = 0;
for ( var i = 2; i < 5; i++ ) {
    http.get( process.argv[ i ], function() {

        return function( response ) {
            response.pipe( bl( function( err, data ) {
                if ( err )
                    return console.error( "Error: "
                ret[    i - 2 ] = data.toString();
                count++;

                if ( count == 3 ) {
                    for ( var j = 0; j < 3; j++ )
                        console.log( ret[ j ] );
                }
            } ) );
        }
    }() );
}
```

To save the current value of i, we need a **closure**.
"Closures are functions that refer to independent (free) variables. In other words, the function defined in the closure 'remembers' the environment in which it was created."

nodejs/ex9-juggling-async-right.js

Exercise 9: Juggling async

Right Solution

Modifying from the wrong solution…

```javascript
var http = require( 'http' );
var bl = require( 'bl' );
var ret = [];
var count = 0;
for ( var i = 2; i < 5; i++ ) {
    http.get( process.argv[ i ], function() {

        return function( response ) {
            response.pipe( bl( function( err, data ) {
                if ( err )
                    return console.error( "Error: " + err );
                ret[    i - 2 ] = data.toString();
                count++;

                if ( count == 3 ) {
                    for ( var j = 0; j < 3; j++ )
                        console.log( ret[ j ] );
                }
            } ) );
        }
    }() );
}
```

This returned function defines a closure.

nodejs/ex9-juggling-async-right.js

Exercise 9: Juggling async

Right Solution

Modifying from the wrong solution...

```javascript
var http = require( 'http' );
var bl = require( 'bl' );
var ret = [];
var count = 0;
for ( var i = 2; i < 5; i++ ) {
    http.get( process.argv[ i ], function() {
        var my_i = i;
        return function( response ) {
            response.pipe( bl( function( err, data ) {
                if ( err )
                    return console.error( "Error: " + err );
                ret[ my_i - 2 ] = data.toString();
                count++;

                if ( count == 3 ) {
                    for ( var j = 0; j < 3; j++ )
                        console.log( ret[ j ] );
                }
            } ) );
        }();
    );
}
```

Save and use the current value of *i* inside the closure.

nodejs/ex9-juggling-async-right.js

Exercise 9: Juggling async

Right Solution

```javascript
var http = require( 'http' );
var bl = require( 'bl' );
var ret = [];
var count = 0;
for ( var i = 2; i < 5; i++ ) {
    http.get( process.argv[ i ], function() {
        var my_i = i;
        return function( response ) {
            response.pipe( bl( function( err, data ) {
                if ( err )
                    return console.error( "Error: " + err );
                ret[ my_i - 2 ] = data.toString();
                count++;

                if ( count == 3 ) {
                    for ( var j = 0; j < 3; j++ )
                        console.log( ret[ j ] );
                }
            } ) );
        }
    }() );
}
```

nodejs/ex9-juggling-async-right.js

Exercise 10: Time server

- **Problem:** Write a TCP time server!
  - Your server should listen to **TCP connections** on the port provided by the first argument to your program
  - For each connection you must write the **current date & 24 hour time** in the format: "**YYYY-MM-DD hh:mm**", followed by a newline character
  - Month, day, hour and minute must be zero-filled to 2 integers
    - For example: "2013-07-06 17:42"

- This exercise demonstrates the power of Node.js!
  - Challenge to CSCI 4430 students: Solve this problem in **C/C++ socket programming**!

Exercise 10: Time server

- To create a **raw TCP server**, use the **net** module
  - Use the method named **net.createServer()**
    - It returns an instance of your server
    - To start listening on a particular port, use **server.listen(*<port>*)**
    - It takes a callback function with the following signature:

```
function callback ( socket ) { /* ... */ }
```

  - The **socket** object passed into the callback function contains a lot of metadata regarding the connection
  - To write data to the **socket**: `socket.write( data );`
  - To close the **socket**: `socket.end();`
  - **Ref.:** http://nodejs.org/api/net.html

  Can be combined

  `socket.end( data );`

Exercise 10: Time server

- To create the date, you will need to create a custom format from a **new Date()** object
- The following methods will be useful:
  - `date.getFullYear()`
  - `date.getMonth()`   `// starts at 0`
  - `date.getDate()`   `// returns the day of month`
  - `date.getHours()`
  - `date.getMinutes()`

Exercise 10: Time server

Solution

```
var net = require( 'net' );
var server = net.createServer( function( socket ) {
    var date = new Date();
    var zerofill = function( val ) {
        return ( val <= 9 ? '0' : '' ) + val;
    };

    socket.end( date.getFullYear() + '-'
                + zerofill( date.getMonth() + 1 ) + '-'
                + zerofill( date.getDate() ) + ' '
                + zerofill( date.getHours() ) + ':'
                + zerofill( date.getMinutes() ) + '\n' );

} );
server.listen( Number( process.argv[ 2 ] ) );
```

nodejs/ex10-time-server.js

Exercise 11: HTTP file server

- *Now we are ready to learn how to use Node.js to implement server-side program!*

- **Problem:** Write an **HTTP server** that **serves the same text file** for each request it receives
  - *1st argument*: **Port number** that the server listens on
  - *2nd argument*: The **location** of the file to serve

- You must use the `fs.createReadStream()` method to stream the file contents to the response
  - It creates a **stream** representing the file
  - Use `src.pipe( dst )` to pipe data from the **src** stream to the **dst** stream

Exercise 11: HTTP file server

- Use the **http** module to create an HTTP server
  - **http.createServer()** take a callback that is called once for each connection received by your server

    ```
    function callback ( request, response ) { /* ... */ }
    ```

  - The two arguments are **Node stream** objects representing the HTTP request and the corresponding response
    - Request is used for **fetch properties**, e.g., the header and query string
    - Response is for **sending data to the client**, both headers and body
  - *Ref.:* http://nodejs.org/api/http.html

Exercise 11: HTTP file server

Solution

```
var http = require( 'http' );
var fs = require( 'fs' );

var server = http.createServer( function( request, response ) {
    response.writeHead( 200, { 'Content-Type' : 'text/plain' } );
    fs.createReadStream( process.argv[ 3 ] ).pipe( response );
} );
server.listen( Number( process.argv[ 2 ] ) );
```

nodejs/ex11-http-file-server.js

Exercise 11: HTTP file server

Solution

```
var http = require( 'http' );
var fs = require( 'fs' );

var server = http.createServer( function( request, response ) {
    response.writeHead( 200, { 'Content-Type' : 'text/plain' } );
    fs.createReadStream( process.argv[ 3 ] ).pipe( response );
} );
server.listen( Number( process.argv[ 2 ] ) );
```

nodejs/ex11-http-file-server.js

Write the status code and headers using the `response.writeHead()` method.

Pipe the data from the stream representing the file to the `response` stream.

Exercise 12: HTTP uppercaserer

- **Problem:** Write an HTTP server that **receives only POST requests** and converts incoming **POST body characters** to **upper-case** and returns it to the client

  - *1st argument*: **Port number** that the server listens on

- You can use the "**through2-map**" module to create a **transform stream** using only a single function that takes a **chunk of data** and returns a chunk of data

  - Install **through2-map** using **npm**

  - Read https://www.npmjs.com/package/through2-map for its usage

Exercise 12: HTTP uppercaserer

Solution

```
var http = require( 'http' );
var map = require( 'through2-map' );

var server = http.createServer( function( request, response ) {
    if ( request.method != 'POST' )
        return response.end( 'Only POST requests are accepted' );

    request
        .pipe( map( function( chunk ) {
            return chunk.toString().toUpperCase();
        } ) )
        .pipe( response );
} );
server.listen( Number( process.argv[ 2 ] ) );
```

nodejs/ex12-http-uppercaserer.js

Exercise 12: HTTP uppercaserer

Solution

```
var http = require( 'http' );
var map = require( 'through2-map' );

var server = http.createServer( function( request, response ) {
    if ( request.method != 'POST' )
        return response.end( 'Only POST requests are accepted' );

    request
        .pipe( map( function( chunk ) {
            return chunk.toString().toUpperCase();
        } ) )
        .pipe( response );
} );
server.listen( Number( process.argv[ 2 ] ) );
```

nodejs/ex12-http-uppercaserer.js

Check is the request is a **POST** request.

Pipe the data from the **request** stream to the **transform stream**.

Pipe the data from the **transform stream** to the **response** stream.

Exercise 13: HTTP JSON API server

- **Problem:** Write an HTTP server that serves **JSON data** when it
  - Receives a GET request to the path "**/api/parsetime**"
    - The JSON response should contain only '**hour**', '**minute**' and '**second**' properties
  - Receives a GET request to the path "**/api/unixtime**"
    - The JSON response should contain the **UNIX epoch time in milliseconds** (the number of milliseconds since 1 Jan 1970 00:00:00 UTC) under the property '**unixtime**'
  - Both requests accept a query string with a key '**iso**' and an ISO-format time as the value
  - *1ˢᵗ argument of the program*: **Port number** that the server listens on

Exercise 13: HTTP JSON API server

- Use the **`url.parse()`** method in the **`url`** module to parse the URL and query string
  - *Ref.:* http://nodejs.org/api/url.html
- Use **`JSON.stringify()`** to convert an object into JSON string format
- To parse a date in **ISO format**, use **`new Date(`** *`<ISO date string>`* **`)`**
- Use **`date.getTime()`** to get the **UNIX epoch time in milliseconds**

Exercise 13: HTTP JSON API server

Solution

```
var http = require( 'http' );
var url = require( 'url' );

var server = http.createServer( function( request, response ) {
    var parsedUrl = url.parse( request.url, true );
    var ret = {};

    switch( parsedUrl.pathname ) {
        case '/api/parsetime':
            var date = new Date( parsedUrl.query.iso );
            ret.hour = date.getHours();
            ret.minute = date.getMinutes();
            ret.second = date.getSeconds();
            break;
```

Check the URL to determine the endpoints.

Read the **iso** property in the query string.

nodejs/ex13-http-json-api-server.js

Exercise 13: HTTP JSON API server

Solution

```
(Continued)
        case '/api/unixtime':
            var date = new Date( parsedUrl.query.iso );
            ret.unixtime = date.getTime();
            break;
        default:
            return response.writeHead( 404 ).end();
    }

    response.writeHead( 200, { 'Content-Type' : 'application/json' } );
    response.end( JSON.stringify( ret ) );
} );
server.listen( Number( process.argv[ 2 ] ) );
```

nodejs/ex13-http-json-api-server.js

Congratulations!

- You have learnt the fundamental concepts involved in Node.js development!

- To develop web applications even faster, we will use a **web framework** called **Express**
  - Please refer to the corresponding tutorial slides

– End –