- Event driven, non-blocking I/O model -> lightweight and efficient
- Node.js applications written in JS

```
Node.js = Runtime Environment + JavaScript libraries
```

```
let calc = require("mylib.js")
```

User-defined Modules

```
// Filename: calc.js
exports.add = function (x, y) {
    return x + y;
};

exports.sub = function (x, y) {
    return x - y;
};

exports.mult = function (x, y) {
    return x * y;
};

// Filename: index.js
var calculator = require('./calc');

var x = 50, y = 20;

console.log("Addition of 50 and 10 is "
                  + calculator.add(x, y));

console.log("Subtraction of 50 and 10 is "
                  + calculator.sub(x, y));

console.log("Multiplication of 50 and 10 is "
                  + calculator.mult(x, y));
```

Buffers

Method	Description	Syntax	Returns	Exam
`Buffer.alloc()`	Creates a new buffer of specified size, initialized to zero.	`Buffer.alloc(size, fill, encoding)`	A new buffer filled with specified values.	`const`

Method	Description	Syntax	Returns	Exam
`Buffer.from()`	Creates a buffer initialized with data (string, array, or another buffer).	`Buffer.from(data, encoding)`	A new buffer with the initialized data.	`const`
`Buffer.allocUnsafe()`	Creates a buffer of specified size without initializing it (may contain old memory).	`Buffer.allocUnsafe(size)`	A new uninitialized buffer.	`const conso`
`buf.write()`	Writes a string to the buffer starting at a specified offset.	`buf.write(string, offset, length, encoding)`	Number of bytes written.	`const conso`
`buf.toString()`	Converts the buffer's contents into a string using specified encoding.	`buf.toString(encoding, start, end)`	A string representation of the buffer's data.	`const conso`
`Buffer.isBuffer()`	Checks if the given object is a buffer.	`Buffer.isBuffer(object)`	`true` if the object is a buffer, `false` otherwise.	`conso // tr`
`buf.length`	Returns the size (in bytes) of the buffer.	`buf.length`	Length of the buffer in bytes.	`const conso`
`buf.copy()`	Copies data from one buffer to another.	`buf.copy(targetBuffer, targetStart, sourceStart, sourceEnd)`	Number of bytes copied.	`const Buffe conso`
`Buffer.concat()`	Concatenates multiple buffers into one.	`Buffer.concat(list, totalLength)`	A new buffer containing the concatenated data.	`const Buffe Buffe`
`buf.slice()`	Returns a shallow copy of a portion of the buffer.	`buf.slice(start, end)`	A new buffer containing the sliced data.	`const buf.s`
`buf.compare()`	Compares the buffer with another buffer.	`buf.compare(target, targetStart, targetEnd, sourceStart, sourceEnd)`	`0` if equal, `1` if the target buffer comes	`const Buffe`

Method	Description	Syntax	Returns	Exam
			first, `-1` otherwise.	
`Buffer.isEncoding()`	Checks if the given encoding is a valid character encoding.	`Buffer.isEncoding(encoding)`	`true` if valid encoding, `false` otherwise.	`conso`
`buf.fill()`	Fills the buffer with a specified value.	`buf.fill(value, offset, end, encoding)`	The modified buffer.	`const conso`

Detailed Examples

1. Creating Buffers

```
// Create a zero-initialized buffer of size 10
const buf1 = Buffer.alloc(10);
console.log(buf1); // <Buffer 00 00 00 00 00 00 00 00 00 00>

// Create a buffer from a string
const buf2 = Buffer.from('Hello');
console.log(buf2.toString()); // Hello
```

2. Reading and Writing

```
const buf = Buffer.alloc(10);

// Write to the buffer
buf.write('NodeJS');
console.log(buf.toString()); // NodeJS

// Read specific parts
console.log(buf.toString('utf8', 0, 4)); // Node
```

3. Copying Buffers

```
const src = Buffer.from('ABC');
const dest = Buffer.alloc(3);

// Copy contents
src.copy(dest);
console.log(dest.toString()); // ABC
```

4. Concatenating Buffers

```
const buf1 = Buffer.from('Hello ');
const buf2 = Buffer.from('World');
const result = Buffer.concat([buf1, buf2]);
console.log(result.toString()); // Hello World
```

5. Checking Buffers

```
const buf = Buffer.from('Hello');
console.log(Buffer.isBuffer(buf)); // true
console.log(buf.length); // 5
```

File System Module (fs)

`fs.open(path, flag[, mode], callback)`

Different Flags

Mode	Description	Behavior
`r`	Open file for **reading** only.	The file must exist; otherwise, an error will be thrown.
`r+`	Open file for **reading and writing**.	The file must exist; otherwise, an error will be thrown.
`rs`	Open file for **reading** in **synchronous** mode.	This bypasses the OS cache and reads directly from the disk. Use sparingly as it can be slower. The file must exist.
`rs+`	Open file for **reading and writing** in **synchronous** mode.	Same as `rs`, but allows both reading and writing. The file must exist.
`w`	Open file for **writing**.	Creates the file if it does not exist. Truncates (empties) the file if it exists.
`wx`	Open file for **writing**, but **fails if the file exists** (exclusive mode).	Ensures the file does not already exist before writing to it.
`w+`	Open file for **reading and writing**.	Creates the file if it does not exist. Truncates (empties) the file if it exists.
`wx+`	Open file for **reading and writing**, but **fails if the file exists** (exclusive mode).	Ensures the file does not already exist before writing to it.
`a`	Open file for **appending**.	Creates the file if it does not exist. Data is added to the end of the file without truncating its content.
`ax`	Open file for **appending**, but **fails if the file exists** (exclusive mode).	Ensures the file does not already exist before appending to it.
`a+`	Open file for **reading and appending**.	Creates the file if it does not exist. Data is added to the end of the file without truncating its content.

Mode	Description	Behavior
`ax+`	Open file for **reading and appending**, but **fails if the file exists** (exclusive mode).	Ensures the file does not already exist before appending to it.

```
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
      return console.error(err);
   }
   console.log("File open successfully");
});
```

fs.writeFile(path, data[,options], callback)

- data -> String/Buffer
- Callback -> receives error as a parameter

```
var fs = require("fs");

console.log("writing into existing file");
fs.writeFile('input.txt', 'Web tech', function(err) {       // write data to a file
   if (err) {
      return console.error(err);
   }

   console.log("Data written successfully!");
   console.log("Let's read newly written data");

   fs.readFile('input.txt', function (err, data) {           // read data to a file
      if (err) {
         return console.error(err);
      }
      console.log("Asynchronous read: " + data.toString());
   });
});

fs.appendFile('input.txt', 'Hello content!', function (err) {
  if (err) throw err;
  console.log('Saved!');
});

fs.open('./input.txt', 'r', (err, fd) => {
        if (err) {
          return console.error(err);
        }

        // Allocate a buffer to hold the read data
        let buf = Buffer.alloc(20);

        // Read data from the file
```

```
        fs.read(fd, buf, 2,  buf.length-2, 0, (err, bytesRead, buffer) => {
          if (err) {
                return console.error(err);
          }

          console.log(`Bytes read: ${bytesRead}`);
          console.log(`Buffer content: ${buffer.toString()} / ${buffer}`);

          // Always close the file descriptor after finishing
          fs.close(fd, (err) => {
                if (err) {
                  return console.error(err);
                }
                console.log('File closed successfully.');
          });
        });
});
```

```
PS C:\Users\appoo\PES\Sem 3\WT\node_testing> node index.js
writing into existing file
Bytes read: 8
Bytes read: 8
Buffer content: Web tech / Web tech
Data written successfully!
Data written successfully!
Let's read newly written data
File closed successfully.
Saved!
Asynchronous read: Web techHello content!
PS C:\Users\appoo\PES\Sem 3\WT\node_testing> node index.js
writing into existing file
Bytes read: 18
Buffer content: Web techHello cont / Web techHello cont
Saved!
Data written successfully!
Let's read newly written data
File closed successfully.
Asynchronous read: Web techHello content!
PS C:\Users\appoo\PES\Sem 3\WT\node_testing>
```

Unlinking a File

Use fs.unlink() method to delete an existing file.
fs.unlink(path, callback);

Closing a File

fs.close(fd, callback)
fd − This is the file descriptor returned by file fs.open() method.
callback − This is the callback function No arguments other than a possible exception are given to the completion callback.

Truncate a File

fs.truncate(fd, len, callback)
fd − This is the file descriptor returned by fs.open().
len − This is the length of the file after which the file will be truncated.
callback − This is the callback function No arguments other than a possible exception are given to the completion callback.

HTTP Module

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

var server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/html' });

  // Writing basic information
  res.write("Hello World!<br><br>");
  res.write(`Request URL: ${req.url}<br>`);

  // Parsing the URL
  var q = url.parse(req.url, true);

  // Writing individual URL components
  res.write(`Full path: ${q.path}<br>`);
  res.write(`Pathname: ${JSON.stringify(q.pathname)}<br>`);

  // Converting query object to string
  res.write(`Query: ${JSON.stringify(q.query)}<br>`);

  // Writing search string (if available)
  if (q.search) {
    res.write(`Search: ${q.search}<br>`);
  } else {
    res.write(`Search: No query string provided<br>`);
  }

  // End the response
  res.end();
}).listen(1030);

console.log("Server up at port 1030!");
```

Validator

```
import val from 'validator';
var email='xyz@pes.edu'
console.log(val.isEmail(email))
email='xyz@.edu'
console.log(val.isEmail(email))
var name='john'
console.log(val.isLowercase(name))
name='JOHN'
console.log(val.isLowercase(name))
var name=''
console.log(val.isEmpty(name))
name='Smith'
console.log(val.isEmpty(name))
```