

## Web Server

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

Node was designed with back end development needs as a top priority. One of these needs is the ability to create web servers. A web server is a computer process that listens for requests from clients and returns responses. A Node core module designed to meet these needs is the http module. This module has functions that simplify receiving and responding to requests.

## Node.js Modules

```
const baseball = require('./babeRuth.js')
```

In Node.js files are called modules. Modularity is a technique where one program has distinct parts each providing a single piece of the overall functionality - like pieces of a puzzle coming together to complete a picture. `require()` is a function used to bring one module into another.

## Asynchronous Node.js

```
let endgame = () => {  
  console.log('I am inevitable')  
};  
  
// endgame will run after 1000ms  
setTimeout(endgame, 1000);
```

Node.js is a non-blocking, asynchronous environment. The **event loop** in Node.js enables asynchronous actions to be handled in a non-blocking way. Node.js provides APIs which allow operations to be put in a queue, waiting to be executed after the previous operation finishes. If synchronous tasks never end, operations waiting in the event-queue will never execute.

## Creating A Server

```
// required in the http core module.
const http = require('http');

let requestListener = (request, response) => {
  // code to be filled in depending on server
};

// assigning return value
const server = http.createServer(requestListener);

// assigning server port
server.listen(3000);
```

`http.createServer()` is a method that returns an instance of an `http.server`. The method `.listen()` in `http.server` tells the server to “listen” for incoming connections. We give `http.createServer()` a callback function also known as the `requestListener`, which will be triggered once the server is listening and receives a request. The requestlistener requests a request object and a response object.

## Readable/Writable Streams

```
// Readable stream
readline.createInterface();

// Writable Stream
fs.createWriteStream();
```

In most cases, data isn't processed all at once but rather piece by piece. This is what we call streams. Streaming data is preferred as it doesn't require tons of RAM and doesn't need to have all the data on hand to begin processing it. To read files line-by-line, we can use the `.createInterface()` method from the readline core module. We can write to streams by using the `.createWriteStream()` method.

## Node.js Process Object

```
if (process.env.NODE_ENV === 'development'){
  console.log('Do not deploy!! Do not deploy!!');
}
```

A process is the instance of a computer program that is being executed. Node has a global process object with useful properties. One of these properties is **NODE\_ENV** which can be used in an if/else statement to perform different tasks depending on if the application is in the production or development phase.

## The `node` Command

```
node app.js
```

We can execute Node.js programs in the terminal by typing the `node` command, followed by the name of the file.

The example command above runs `app.js`.

## Node.js REPL

```
//node is typed in the console to access REPL
$ node

//the > indicates that REPL is running
// anything written after > will be evaluated
> console.log("HI")

// REPL has evaluated the line and has printed out HI
HI
```

Node.js comes with REPL, an abbreviation for read-eval-print loop. REPL contains three different states:

\*a **read** state where it reads the input from a user, \*the **eval** state where it evaluates the user's input \*the **print** state where it prints out the evaluation to the console.

After these states are finished REPL loops through these states repeatedly. REPL is useful as it gives back immediate feedback which can be used to perform calculations and develop code.

## Node.js process.argv

```
// Command line values: node web.js testing several features  
console.log(process.argv[2]); // 'testing' will be printed
```

`process.argv` is a property that holds an array of command-line values provided when the current process was initiated. The first element in the array is the absolute path to the Node, followed by the path to the file that's running and finally any command-line arguments provided when the process was initiated.

## Node.js process.memoryUsage()

```
//using process.memoryUsage() will return an object in  
a format like this:
```

```
{ rss: 26247168,  
  heapTotal: 5767168,  
  heapUsed: 3573032,  
  external: 8772 }
```

`process.memoryUsage()` is a method that can be used to return information on the CPU demands of the current process. Heap can refer to a specific data structure or to the computer memory.

## The `error` Module

The asynchronous operations involving the Node.js APIs assume that the provided callback functions should have an error passed as the first parameter. If the asynchronous task results in an error, the error will be passed in as the first argument to the callback function. If no error was thrown, then the first argument will be `undefined`.

## Input/Output

```
// Recieves an input
process.stdin.on();

// Gives an output
process.stdout.write();
```

Input is data that is given to the computer, while output is any data or feedback that a computer provides. In Node, we can get input from a user using the `stdin.on()` method on the `process` object. We are able to use this because `.on()` is an instance of `EventEmitter`. To give an output, we can use the `.stdout.write()` method on the process object as well. This is because `console.log()` is a thin wrapper on `.stdout.write()`.

## Node Package Manager

NPM stands for node-package-manager. An NPM is essentially a collection of code from other developers that we can use. When Node is installed the npm command-line tool is downloaded as well. This command-line tool enables us to interact with the registry via our terminal.

## The `fs` Module

```
// First argument is the file path
// The second argument is the file's character encoding
// The third argument is the invoked function
fs.readFile('./file.txt', 'utf-8', CallbackFunction);
```

The *filesystem* controls how data on a computer is stored and retrieved. Node.js provides the `fs` core module, which allows interaction with the filesystem. Each method provided through the module has a synchronous and asynchronous version to allow for flexibility. A method available in the module is the `.readFile()` method that reads data from the provided file.



## Node.js Core Modules

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

Node has several modules included within the environment to efficiently perform common tasks. These are known as the **core modules**. The core modules are defined within Node.js's source and are located in the `lib/` folder. A core module can be accessed by passing a string with the name of the module into the `require()` function.

## Node.js Local Modules

```
// type.js
// by using the export property we can use this module in
another file
module.exports = class key {
  constructor(car) {
    this.car = car;
  }
};

// qwerty.js
// by requiring the type.js file we can use the module in
the type.js file
let Dog = require('./type.js');
```

In Node.js files are considered modules. Modules that are created locally are called local modules. These local modules are held in an object called `module`. This object has a property called `exports` which allows a module to be accessed in a different module.