Introduction to Node.js

Node.js REPL

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 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 Global Object

The Node.js environment has a global object that contains every Node-specific global property. The global object can be accessed by either typing in `console.log(global)` or `global` in the terminal after RPL is running. In order to see just the keys `Object.keys(global)` can be used. Since global is an object, new properties can be assigned to it via `global.name_of_property = 'value_of_property'` .

```
//Two ways to access global
> console.log(global)
//or
> global

//Adding new property to global
> global.car = 'delorean'
```

Node.js Process Object

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.

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

Node.js process.argv

`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 web.js testing several features
console.log(process.argv[2])// 'features' will be printed
```

Node.js process.memoryUsage()

`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.

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

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

Node.js Modules

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.

```
let baseball = require(./babeRuth.js)
```

Node.js Core Modules

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.

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

Node.js Local Modules

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 `export` which allows a module to be accessed in a different module.

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

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.

EventEmitter Class

Node.js has an **EventEmitter** class which can be accessed by requiring the **events** core module. Each event emitter instance has an `.on()` method which assigns a listener callback function to a named event. EventEmitter also has an `.emit()` method which announces a named event that has occurred.

```
// Require in the 'events' core module
let events = require('events');

// Create an instance of the EventEmitter class
let myEmitter = new events.EventEmitter();
let version = (data) => {
 console.log(`participant: ${data}.`);
};

// Assign the version function as the listener callback for
'new user' events
myEmitter.on('new user', version)

// Emit a 'new user' event
myEmitter.emit('new user', 'Lily Pad')
// 'Lily Pad'
```

Asynchronous Node.js

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.

```
let endgame = () => {
  console.log('I am inevitable')
};

// endgame will run after 1000ms
setTimeout(endgame, 1000);
```

Asynchronous Error Handling

The Node environment has all the standard JavaScript errors as well as the JavaScript **Error** class for creating new error instances. Many asynchronous Node APIs use error-first callback functions: callback functions which have an error as the first expected argument and the data as the second argument. If the asynchronous task results in an error, it will be passed in as the first argument to the callback function. If no error was thrown, the first argument will be `undefined` .

Node.js Input/Output

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 by 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()` .

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

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

Filesystem

A filesystem is how you access and organize all data on a computer. The Node.js `fs` core module is an API for interacting with the file system. Each method available through the `fs` module has a synchronous version and an asynchronous version. One of the methods in the `fs` core module is the `.readfile()` method which allows us to read data from a file.

```
// 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);
```

Web Server

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.

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

Creating A Server

`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.

```javascript
// 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);
```

Readable/Writable Streams

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.

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

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

Learn Node-SQLite

sqlite3 Module

Node.js and SQLite are separate entities, but both can be used together to create powerful applications. First, we need to link them together by requiring the **sqlite3** module. Once we have required this module in our JavaScript, it will give us access to methods to perform various actions with the database.

```
//requiring the sqlite3 module

const sqlite3 = require('sqlite3');
```

Creating/Connecting To A Database

In SQLite, a database corresponds to a single file. The `sqlite3.Database()` method can be used to create or connect to a single file.

```
// creating a database file
const db = new sqlite3.Database('./db.sqlite');
```

db.get() Method

Sometimes we only need one row from a database. The method `db.get()` allows us to fetch a single row from a database matching a query. If multiple rows match the query only one row matching the query will be returned.

```
// This will return the first row matching the query.
db.get("SELECT * FROM drinks WHERE type = 'soda'")
```

db.all() Method

A query is a statement that speaks to a database and requests specific information from it. The `db.all()` method allows us to execute a query that returns all rows with specific data. For example, we can use `db.all()` to return all rows from a table that has the pet as a cat.

```
// Selects a table named Animal and returns only the rows
the has the pet as a cat
db.all("SELECT * FROM Animal WHERE pet = 'cat'")
```

db.each() Method

Sometimes we want to perform an action every time a row is returned. Using the `db.each()` method we can do exactly that. `db.each()` takes a query and a callback function that it performs on each row returned from the query. `db.each()` can also take a second callback function, which will be called when all of the queries are completed and processed.

```
db.each("SELECT * FROM Sports WHERE type = 'baseball'",
  (error, row) => {
  // This will be printed everytime a row is returned
  console.log(`${row.name} is a good baseball team`);
};
```

db.run() Method

Sometimes we want to do more than just get a result from a database. The `db.run()` method holds SQL commands that do not return rows; such as commands that will allow us to create, update tables, or insert new rows.

```
//creating a table
db.run("CREATE TABLE user (id INT, dt TEXT)");

//updating existing table
db.run('INSERT INTO Key (type, color) VALUES ($type, $color)
```

db.serialize() Method

Any requests we send to our database get processed as quickly as possible, which can lead to multiple requests processing at the same time. Usually, this creates efficiency but in some cases like when we want to create a table and insert rows, it can cause errors. This is because our request might try to insert a row into a table that's not even created yet. This problem can be solved by the `db.serialize()` method which takes multiple requests and executes them one by one.

```
//each request inside this method will be executed one by
one
db.serialize(() => {
  db.run("DROP TABLE Stocks");
  db.run("CREATE TABLE Stocks");
  db.run("INSERT INTO Stocks (AMD, MSFT, TSLA);
});
```

Handling Errors

sqlite3 uses Node.js error-first callback style.The first argument in the methods `db.run()` , `db.each()` , `db.get()` , and `db.all()` will always be an Error object. This object will return an error if it exists and return `null` if no errors are found.

```
// an if statement can be used to log the error
if (err) {
        console.log(err);
}
```