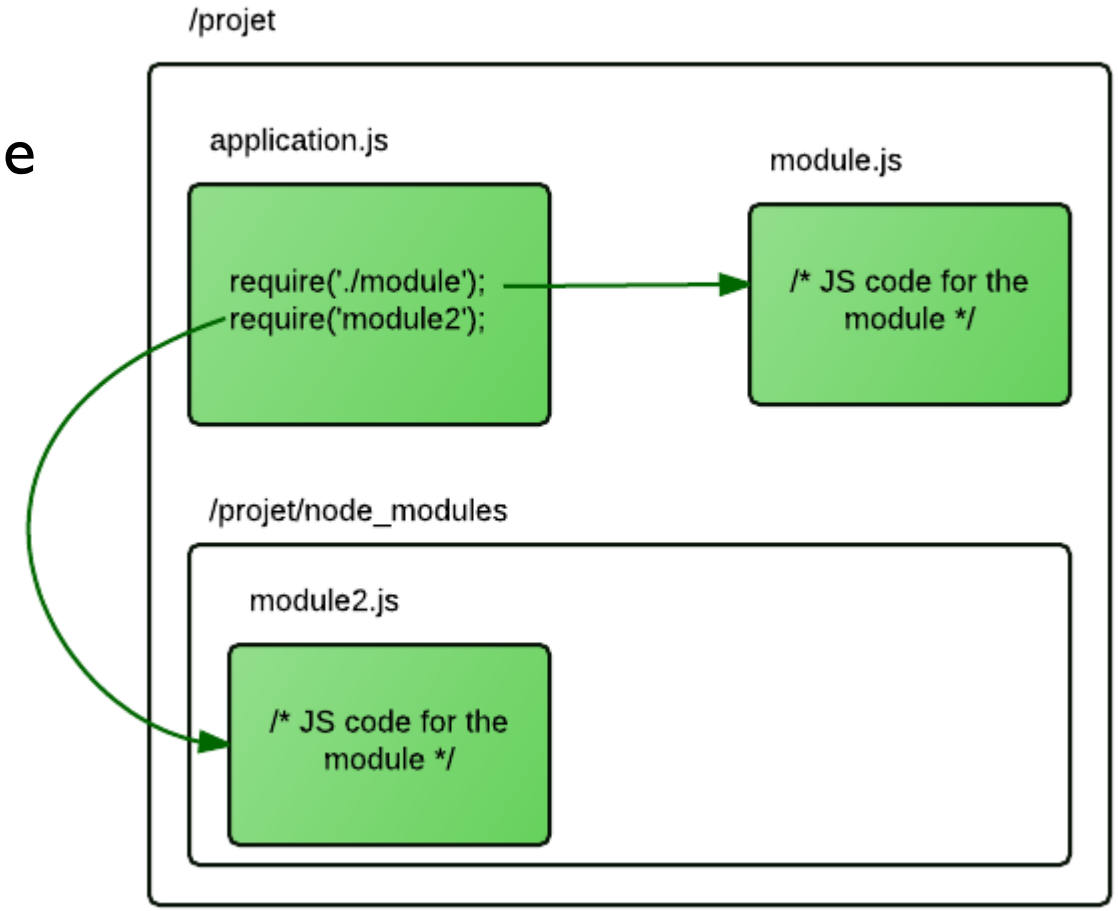
Modules in NodeJS

Rujuan Xing

Modules in NodeJS

- Consider modules to be the same as JavaScript libraries.
- A set of functions you want to include in your application.

- Node implements **CommonJS** Modules specs.
  - CommonJS module are defined in normal `.js` files using `module.exports`
  - In Node.js, each file is treated as a separated module



https://openclassrooms.com

Type of Modules

- Built-in Modules
  - Node.js has a set of built-in modules which you can use without any further installation.
  - buffer, fs, http, path, etc.
  - Built-in Modules Reference
- 3rd party modules on www.npmjs.com
- Your own Modules
  - Simply create a normal JS file it will be a module *(without affecting the rest of other JS files and without messing with the global scope).*

Include Modules – `require()` function

- The basic functionality of `require` is that it reads a JavaScript file, executes the file, and then proceeds to return the `module.exports` object.
  - `const path = require('path');`
  - `const config = require('./config');`

- Rules:
  - If the file doesn't start with "`./`" or "`/`", then it is either considered a **core module** (and the local Node path is checked), or a dependency in the local **node_modules** folder.
  - If the file starts with "`./`" it is considered a relative file.
  - If the file starts with "`/`", it is considered an absolute path.
  - If the filename passed to require is a **directory**, it will first look for `package.json` in the directory and load the file referenced in the main property. Otherwise, it will look for an `index.js`.
  - NOTE: you can omit "`.js`" and `require` will automatically append it if needed.

How `require('/path/to/file')` works

- Node goes through the following sequence of steps:
  1. Resolve: to find the absolute path of the file
  2. Load: to determine the type of the file content
  3. Wrap: to give the file its private scope
  4. Evaluate: This is what the VM does with the loaded code
  5. Cache: when we require this file again, don't go over all the steps.

- **Note**: Node core modules return immediately (no resolve)

What's the wrapper?

- `node -p "require('module').wrapper"`

1. Node will wrap your code into:

```
(function (exports, require, module, __filename, __dirname){
        let exports = module.exports;
    // this is why can use exports and module objects.. etc in your code
    without any problem, because Node is going to initialize these and pass
    them as parameters through this wrapper function
});
```

2. Node will run the function using `.apply()`
3. Node will return the following:

```
return module.exports;
```

module.exports

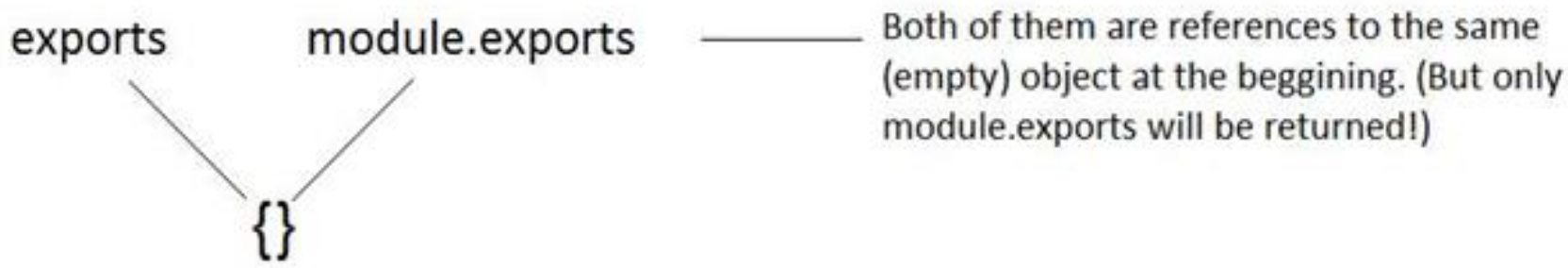
- Think about this object (`module.exports`) as return statement.

```
// helloModule.js
let sayHi = function(){
                console.log('hi');
            }

module.exports = sayHi;
```

```
// app.js
let hello = require('./helloModule');

hello();
```

exports vs module.exports

- `exports` object is a reference to the `module.exports`, that is shorter to type



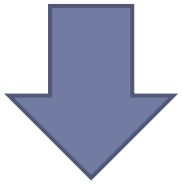
- Be careful when using `exports`, a code like this will make it point to another object. At the end, `module.exports` will be returned.

```
exports = function doSmething() {
    console.log('blah blah');
}
```

doSomething() isn't exported.

Using Modules – Pattern 1

```
// Pattern1 - pattern1.js
module.exports = function () {
    console.log('Josh Edward');
};
```



```
// app.js
const getName = require('./pattern1');
getName(); // Josh Edward
```

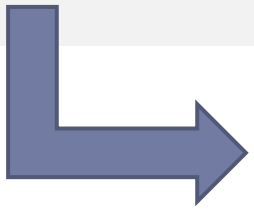
Using Modules – Pattern 2

```
// Pattern2 - pattern2.js
module.exports.getName = function () {
    console.log('Josh Edward');
};

// OR
exports.getName = function () {
    console.log('Josh Edward');
};
```

```
// app.js
const getName = require('./pattern2').getName;
getName(); // Josh Edward

// OR
const person = require('./pattern2');
person.getName(); // Josh Edward
```

Using Modules – Pattern 3

```
// Pattern - pattern4.js
class Person {
    constructor(name = 'Josh Edward') {
        this.name = name;
    }

    getName() {
        console.log(this.name);
    }
}

module.exports = Person;
```



```
// app.js
const Person = require('./pattern4');
const personObj1 = new Person();
personObj1.getName() // Josh Edward
personObj1.name = 'Emma Smith';
personObj1.getName(); //Emma Smith

const Person2 = require('./pattern4');
const personObj2 = new Person2();
personObj2.getName(); // Josh Edward
```

Using Modules – Pattern 4

```
// Pattern5 - pattern5.js
const name = 'Josh Edward';
function getName() {
    console.log(name);
}
module.exports = {
    getName: getName // closure
}
```

```
// app.js
const getName = require('./pattern5').getName;
getName(); // Josh Edward
```

Node core libraries

- Node provides many core libraries

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

fs.readFile('hello.txt', () => console.log('this is readFile 1'));
```

- Read API documentation

Review: Binary data

- Binary is simply a set or a collection of 1s and 0s.
  - 10, 01, 001, 1110, 00101011
  - Each number in a binary, each 1 and 0 in a set are called a **Bit**, which is a short form of **Binary digIT**.
- Example:
  - Numbers: to Store number 12, convert 12 to its binary representation 1100
  - How about strings, images, videos?
    - Computers know how to represent all types of data in binaries.
    - String: To store any character in binaries, computers will first convert the character to a number, then convert that number to its binary representation. The number representation is called Character Code or Code Point.
      - "L".charCodeAt(0) -> 76

Review: Character Set

- How does a computer know what number represent each character?
- A character set is a collection of characters, symbols, and glyphs that are used to represent text in a computing system. It defines a specific mapping between numerical codes and characters, assigning a unique code to each character in the set.
  - Different definitions of these rules: Unicode, ASCII
- So does computer just convert 76 to the base-2 numeral system?

Review: Character Encoding

- There are rules that define how that number should be represented in binaries. Specifically, how many bits to use to represent the number. **- Character Encoding**
- UTF-8: states that characters should be encoded in bytes. A byte is a set of eight bits. Eight 1s and 0s should be used to represent the Code Point of any character in binary.
  - 12 -> 1100 -> UTF: 00001100
  - L -> 76 -> UTF: 01001100

- character set vs character encoding
  - the character set is the actual collection of characters, while character encoding is the method used to represent those characters as binary data.
  - In other words, a character set is a list of characters, and character encoding is a way to encode those characters into binary data.

Stream

- A sequence of data being moved from one point to the other over time.
  - Why stream?
    - You have a huge amount of data to process, but you don't need to wait for all the data to be available before you start processing it.
  - What is stream?
    - The big data is broken down and sent in chunks, then each chunk is being moved in the file system.
      - Move the texts stored in file1.txt to file2.txt
  - How?

Buffer Intro

- Typically, the movement of data is usually with the intention to process it, or read it, and make decisions based on it. But there is a minimum and a maximum amount of data a process could take over time. So if the rate the data arrives is faster than the rate the process consumes the data, the excess data need to wait somewhere for its turn to be processed.

- On the other hand, if the process is consuming the data faster than it arrives, the few data that arrive earlier need to wait for a certain amount of data to arrive before being sent out for processing.

- That "waiting area" is the buffer! It is a small physical location in your computer, usually in the RAM, where data are temporally gathered, wait, and are eventually sent out for processing during streaming.

Buffer

- A buffer is an area of memory. It represents a fixed-size chunk of memory (can't be resized) allocated outside of the V8 JavaScript engine.
  - Node.js can’t control the speed or time of data arrival, the speed of the stream. It only can decide when it’s time to send out the data. If it’s not yet time, Node.js will put them in the buffer — the “waiting area” — a small location in the RAM, until it’s time to send them out for processing.
- while in the buffer, we can manipulate or interact with the binary data being streamed.

Buffer Example

```
const buf = Buffer.from('Hey!'); //create a buffer
//Those numbers are the UTF-8 bytes that identify the characters in the buffer (H → 72, e → 101, y → 121).
// This happens because Buffer.from() uses UTF-8 by default.
console.log(buf[0]); //72
console.log(buf[1]); //101
console.log(buf[2]); //121

console.log(buf.toString());
console.log(buf.length);
for (const item of buf) {
    console.log(item); //72 101 121 33
}
```

`path` module

- The `path` module provides a lot of very useful functionality to access and interact with the file system.

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

//Return the directory part of a path:
console.log(path.dirname('Buffer'));

//Joins two or more parts of a path:
const name = 'joe';
console.log(path.join('users', name, 'notes.txt'));
```

`fs` module

- The `fs` module provides a lot of very useful functionality to access and interact with the file system.

```
const fs = require('fs');
const path = require('path');
console.log(__dirname); // returns absolute path of current file
const greet = fs.readFileSync(path.join(__dirname, 'greet.txt'), 'utf8');
console.log(greet);

const greet2 = fs.readFile(path.join(__dirname, 'greet.txt'), 'utf8',
    function(err, data) { console.log(data); });
console.log('Done!');
// Hello
// Done!
// Hello
```

- Notice the Node Applications Design: any async function accepts a **callback as a last parameter** and the **callback function accepts error as a first parameter** (`null` will be passed if there is no error).
- Notice: `data` here is a buffer object. We can convert it with `toString` or add the encoding – `'utf8'`

Example Read/Write Files

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

// Reading from a file:
fs.readFile(path.join(__dirname, 'greet.txt'), { encoding: 'utf8' }, (err, data) => {
    if (err) throw err;
    console.log(data);
});

// Writing to a file:
fs.writeFile('students.txt', 'Hello World!', (err) => {
    if (err) throw err;
    console.log('Done');
});
```

What's the problem with the code above?

Stream

- Stream is a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way.

- Why streams?
  - Memory efficiency: you don't need to load large amounts of data in memory before you are able to process it
  - Time efficiency: it takes way less time to start processing data, since you can start processing as soon as you have it, rather than waiting till the whole data payload is available

- The Node.js stream module provides the foundation upon which all streaming APIs are built. All streams are instances of EventEmitter

Different types of streams

- Readable: a stream you can pipe from, but not pipe into (you can receive data, but not send data to it). When you push data into a readable stream, it is buffered, until a consumer starts to read the data. (`fs.createReadStream`)
- Writable: a stream you can pipe into, but not pipe from (you can send data, but not receive from it). (`fs.createWriteStream`)
- Duplex: a stream you can both pipe into and pipe from, basically a combination of a Readable and Writable stream. (`net.Socket`)
- Transform: a Transform stream is similar to a Duplex, but the output is a transform of its input. (`zlib.createGzip`)

Examples of Readable and Writable streams

Readable Streams

- HTTP responses, on the client
- HTTP requests, on the server
- fs read streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdout and stderr
- process.stdin

Writable Streams

- HTTP requests, on the client
- HTTP responses, on the server
- fs write streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdin
- process.stdout, process.stderr

Stream example

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

// Stream will read the file in chunks
// if file size is bigger than the chunk then it will read a chunk and emit a 'data' e
vent.
// Use encoding to convert data to String of hex
// Use highWaterMark to set the size of the chunk. Default is 64 kb

const readable = fs.createReadStream(path.join(__dirname, 'card.jpg'),
       { highWaterMark: 16 * 1024 });

const writable = fs.createWriteStream(path.join(__dirname, 'destinationFile.jpg'));

readable.on('data', function(chunk) {
    console.log(chunk.length);
    writable.write(chunk);
});
```

Pipes: src.pipe(dst);

- To connect two streams, Node provides a method called `pipe()` available on all readable streams. Pipes hide the complexity of listening to the stream events.

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

const readable = fs.createReadStream(path.join(__dirname, 'card.jpg'));
const writable = fs.createWriteStream(path.join(__dirname, 'destinationFile.jpg'));

readable.pipe(writable);

// note that pipe return the destination, this is why you can pipe it again to another
stream if the destination was readable in this case it has to be DUPLEX because you
are going to write to it first, then read it and pipe it again to another writable
stream.
```

Zip file using streams

```
const fs = require('fs');
const zlib = require('zlib');
const path = require('path');

// this is a readable & writable stream and it returns a zipped stream
const gzip = zlib.createGzip();

const readable = fs.createReadStream(path.join(__dirname, 'source.txt'));
const compressed = fs.createWriteStream(path.join(__dirname, 'destination.txt.gz'));

readable.pipe(gzip).pipe(compressed);
```

- A key goal of the stream API, is to limit the buffering of data to acceptable levels such that sources and destinations of differing speeds will not overwhelm the available memory.

Node as a Web Server

- Node started as a Web server and evolved into a much more generalized framework.
- Node `http` module is designed with streaming and low latency in mind.
- Node is very popular today to create and run Web servers.

Web Server Example

```
const http = require('http');
const server = http.createServer();
server.on('request', function(req, res) {
                        res.writeHead(200, {'Content-Type': 'text/plain'});
                        res.write('Hello World!');
                        res.end();
                    });
server.listen(3000);
```

http.IncomingMessage
Implements ReadableStream Interface

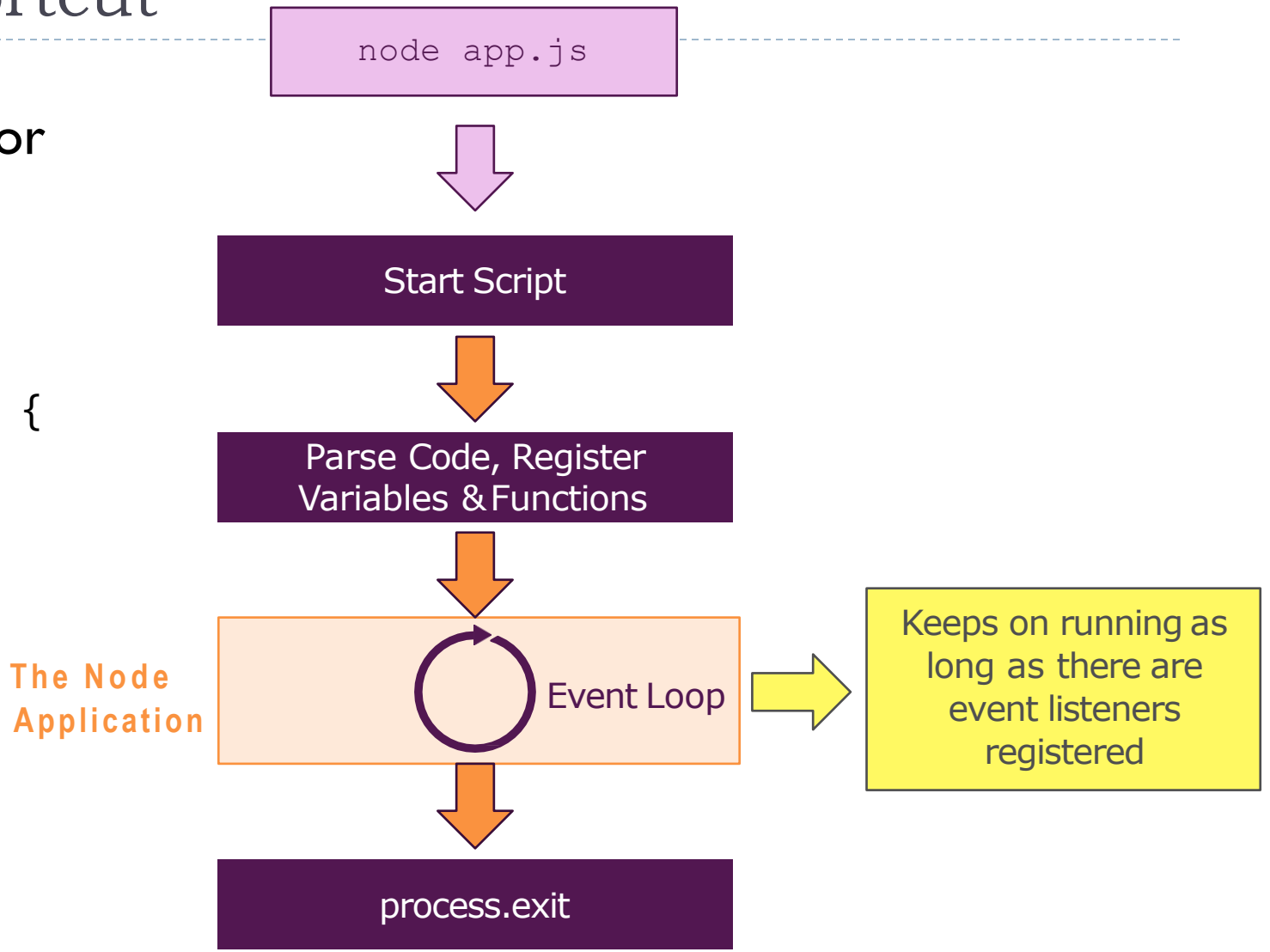
http.ServerResponse
Implements WritableStream Interface

After we run this code. The node program doesn't stop.. it keeps waiting for request

Web Server Example Shortcut

- Passing a callback function to `createServer()` is a shortcut for listening to "request" event.

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

http.createServer(function (req, res) {
    res.writeHead(200, { 'Content-
Type': 'application/json' });
    const person = {
        firstname: 'Josh',
        lastname: 'Edward'
    };
    res.end(JSON.stringify(person));
}).listen(3000, '127.0.0.1');
```



Send out an HTML file

- What's the problem with the code below?

```
index.html
<html>
        <head></head>
        <body>
                <h1>{Message}</h1>
        </body>
</html>
```

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

http.createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    let html = fs.readFileSync(path.join(__dirname, 'index.html'), 'utf8');
    html = html.replace('{Message}', 'Hello from Node.js!');
    res.end(html);
}).listen(3000, '127.0.0.1', () => { console.log('listening on 3000...') });
```

A Simpler solution – Use Stream

- We can simply use stream.pipe(), which does exactly what we described.

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

const server = require('http').createServer();

server.on('request', (req, res) => {
     fs.createReadStream('./big.file').pipe(res);
});

server.listen(8000);
```

Resources

- Node Resources
  - Node Modules
  - Node HTTP
  - Anatomy of an HTTP Transaction
  - fs Module
  - path Module

- Other Resources
  - CommonJS
  - CommonJS Module Format
  - RequireJS
  - Hypertext Transfer Protocol
  - List of HTTP Status Codes
  - Postman
  - https://www.freecodecamp.org/news/do-you-want-a-better-understanding-of-buffer-in-node-js-check-this-out-2e29de2968e8/