



KINGSLAND
UNIVERSITY

Streams and Utilities



Streams, Pub/Sub Pattern, Events, FS Module



Table of Contents

1. Streams
2. Pub/Sub Pattern
3. Events
4. FS Module
5. Debugging





Have a Question?

#js-web



Streams

Streams, Buffers and Chunks

Streams

🕒 Collections of data that is not available at once

- 🕒 Data may come **continuously in chunks**

🕒 Types

- 🕒 **Readable** - can only be read (process.stdin)

- 🕒 **Writeable** - can only be written to (process.stdout)

- 🕒 **Duplex** - both Readable and Writeable (TCP sockets)

- 🕒 **Transform** - the output is computed from the input
(zlib, crypto)

Readable Stream

↳ Functions

- ↳ **read()** - get chunks from the stream
- ↳ **resume()** - switch to **flowing** mode
- ↳ **pause()** - switch to **paused** mode

↳ Events - used when the stream is **flowing**

- ↳ **data** - chunk is available for reading
- ↳ **end** - no more data
- ↳ **error** - an exception has occurred

Readable Stream (2)

HTTP Request is a readable stream

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

http.createServer((req, res) => {
  if (req.method === 'POST') {
    let body = '';
    req.on('data', data => { body += data });
    req.on('end', () => {
      console.log(body);
    });
  }
}).listen(5000);
```

Writable Stream

↳ Functions

↳ **write()** - send chunks to the stream

↳ **end()** - close the stream

↳ Events

↳ **drain** - stream can receive more data

↳ **finish** - all data has been flushed (buffer is empty)

↳ **error** - an exception has occurred

Writable Stream (2)

HTTP Response is a writeable stream

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

server.on('request', (req, res) => {
  const src = fs.createReadStream('./bigfile.txt');
  src.on('data', data => res.write(data));
  src.on('end', () => res.end());
});

server.listen(5000);
```

Piping Streams

- ⌚ The **pipe()** function allows a readable stream to **output directly** to a writable stream
- ⌚ Event listeners are automatically added

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

server.on('request', (req, res) => {
  const src = fs.createReadStream('./bigfile.txt');
  src.pipe(res);
});
server.listen(5000);
```

Duplex and Transform Streams

⌚ Duplex stream

- ⌚ Implements both the **Readable** and **Writable** interfaces
- ⌚ Example - a TCP socket

⌚ Transform stream

- ⌚ A special kind of duplex stream where the output is a **transformed** version of the input
- ⌚ <http://codewinds.com/blog/2013-08-20-nodejs-transform-streams.html>

Streams

🔗 Transforms with Gzip

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

let readStream = fs.createReadStream('index.js');
let writeStream = fs.createWriteStream('index.js.gz');

let gzip = zlib.createGzip();

readStream.pipe(gzip).pipe(writeStream);
```

🔗 https://nodejs.org/dist/latest-v6.x/docs/api/zlib.html#zlib_compressing_http_requests_and_responses

File Upload

⌚ Using **formidable** to upload files

```
let form = new formidable.IncomingForm();

form.parse(req, (err, fields, files) => {
  if (err) {
    console.log(err);
    return;
  }
  console.log(fields);
  console.log(files);
})
```

⌚ Do not forget the **enctype**!



Publish-Subscribe Pattern

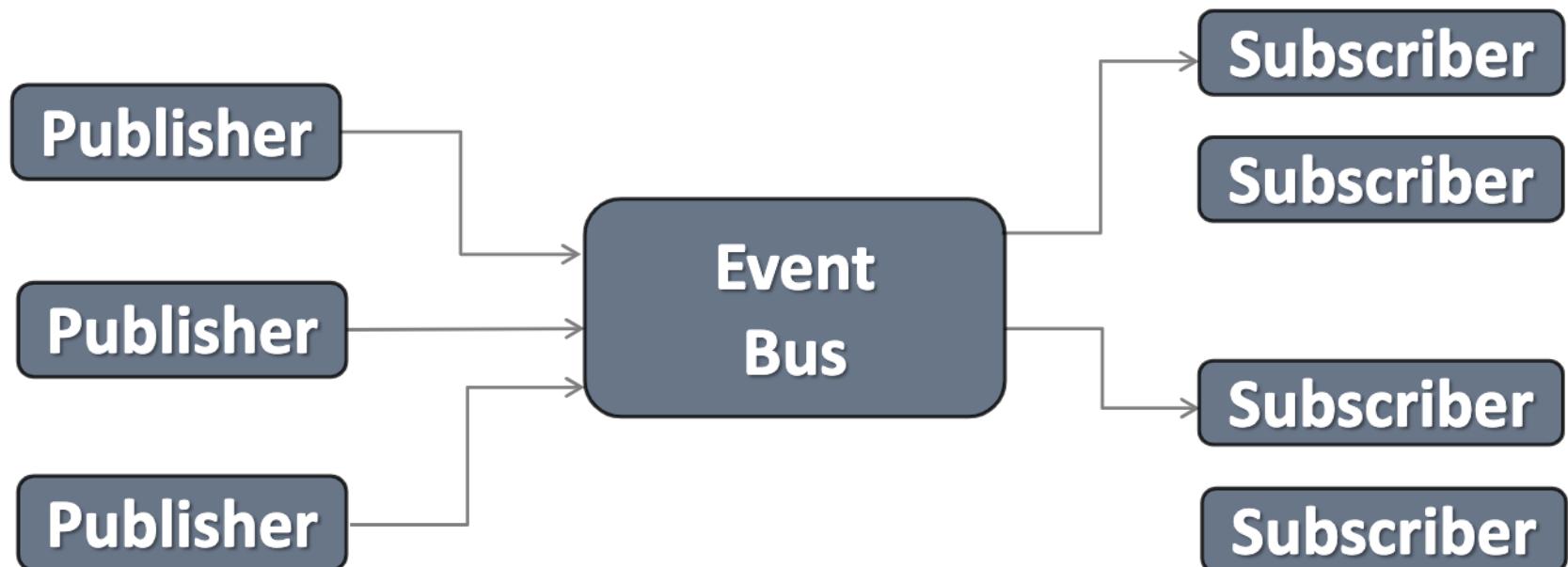
Messaging Pattern

What is Pub/Sub?

- 🕒 Used to **communicate messages** between different system components without them knowing anything about each other's **identity**
 - 🕒 Senders (publishers), do not program the messages to be sent directly to specific **receivers** (subscribers)
 - 🕒 Subscribers express interest in **one or more events**, and only **receive messages** that are of **interest**

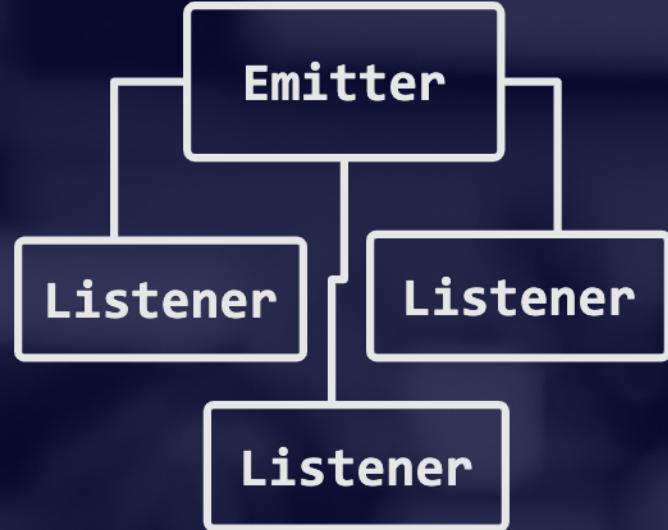
Pub/Sub Example

- 🕒 An intermediary (called a "message broker" or "event bus")
 - 🕒 Receives **published** messages
 - 🕒 Forwards them to the **subscribers** who are registered to receive them



Advantages

- Eliminate Polling
 - Promotes **faster response time and reduces the delivery latency**
- Dynamic Targeting
 - Makes discovery of services easier, more natural and **less error prone**
- Decouple and Scale Independently
 - Makes software more **flexible**
- Simplify Communication
 - Reduces complexity by **removing all the point-to-point connections** with a single connection



Events

Emit Your Data

Events

⌚ Require module "events"

```
const events = require('events');
let eventEmitter = new events.EventEmitter();

eventEmitter.on('click', (a, b) => {
  console.log('A click has been detected!');
  console.log(a + ' ' + b); // outputs 'Hello world'
});

eventEmitter.emit('click', 'Hello', 'world');
```

⌚ Events are **not** asynchronous



FS Module

Working with the File System

Working with the File System

- The `fs` module gives you access to the

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

- All functions have **synchronous** and **asynchronous** variants

```
let data = fs.readFileSync('./package.json', 'utf8');
console.log(data);
```

```
let data = fs.readFile('./package.json', 'utf8',
  (err, data) => { // Handle possible errors
    console.log(data); });

```

Working with the File System (2)

>List files in a directory

```
let data = fs.readdirSync('./myDir', 'utf8');  
console.log(data);
```

```
let data = fs.readdir('./myDir', 'utf8', (err,  
data) => {  
  if (err) {  
    console.log(err);  
    return;  
  }  
  console.log(data);  
});
```

The result is an **array of strings**,
containing all filenames

Working with the File System (3)

Create a directory

```
fs.mkdirSync('./myDir');
```

```
fs.mkdir('./myDir', err => {
  if (err) {
    console.log(err);
    return;
  }
});
```

Working with the File System (4)

Rename file or directory

```
fs.renameSync('./oldName', './newName');
```

```
fs.rename('./oldName', './newName', err => {
  if (err) {
    console.log(err);
    return;
  }
});
```

Working with the File System (5)

Write a file

```
const fs = require('fs');
let filePath = './data.txt';
let data = 'Some text';
```

```
fs.writeFileSync(filePath, data);
```

```
fs.writeFile(filePath, data, err => {
  if (err) {
    console.log(err);
    return;
  }
});
```

Working with the File System (6)

Delete file

```
fs.unlinkSync('./target.txt');
```

```
fs.unlink('./target.txt', err => {
  if (err) {
    console.log(err);
    return;
  }
});
```

Working with the File System (7)

Delete directory

```
fs.rmdirSync('./myDir');
```

```
fs.rmdir('./myDir', err => {
  if (err) {
    console.log(err);
    return;
  }
});
```

Full API docs: <https://nodejs.org/api/fs.html>



Debugging

Inspectors and Watchers



Debugging & Watching in Node.js

⌚ Debugging in Node.js

- ⌚ The V8 **debug protocol** is a **JSON** based protocol

⌚ IDEs with a debugger

- ⌚ Webstorm

- ⌚ Visual Studio

- ⌚ Node-inspector (not working with latest version)

⌚ Watching with **Nodemon**



Live Exercises

Summary

- Node.js has various useful **utility** modules
- **Streams** allow working with **big data**
- **Events** simplify **communication** within a large application
- **Pub/Sub** pattern is used to **communicate messages**
- The **fs** module gives you access to the **file system**





Questions?



License

- ⌚ This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- ⌚ Unauthorized copy, reproduction or use is illegal
- ⌚ Kingsland University – <https://kingslanduniversity.com>





THANK YOU