Streams and Utilities

Streams, Pub/Sub Pattern, Events, FS Module

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

Table of Contents

sli.do

#js-web

Publish-Subscribe 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
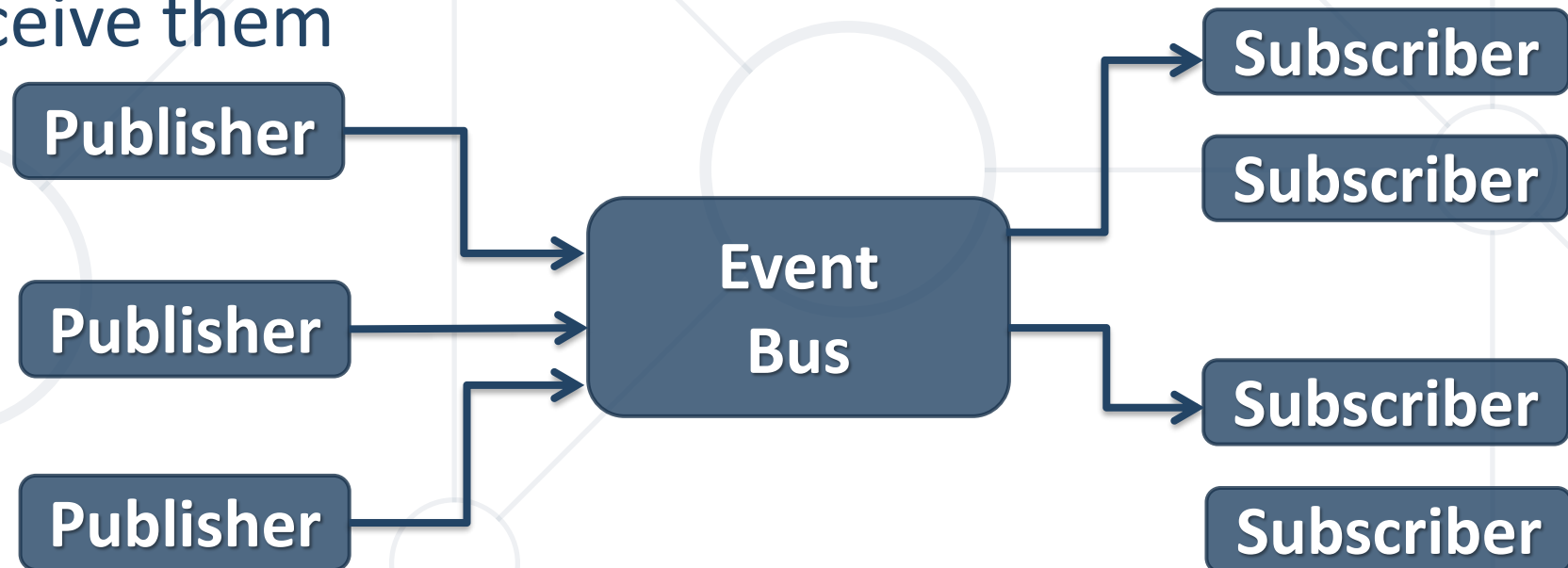
- Decouple and Scale Independently

  - Makes software more **flexible**

- Eliminate Polling

  - Promotes **faster response time** and **reduces the delivery latency**

- Simplify Communication

  - Reduces complexity by **removing** all the **point-to-point connections** with a single connection
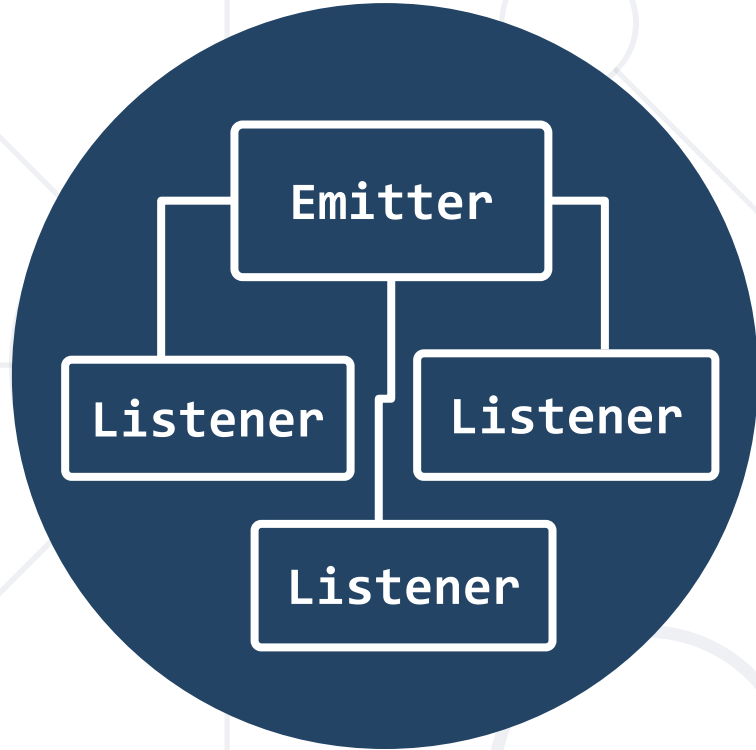
Events

Events

- Require module "**events**"

```javascript
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

Streams

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
  - **pause()** - switch to **paused** mode
  - **resume()** - switch to **flowing** 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

```javascript
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

```javascript
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

```javascript
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 **Writeable** 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/api/zlib.html

FS Module

Working with the File System

- The **fs** module gives you access to the **file system**

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

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

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

```
const 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

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

```javascript
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)

- **Delete** directory

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

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

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

Working with the File System (5)

- **Rename** file or directory

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

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

Working with the File System (6)

- **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 (7)

- **Delete** file

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

```
fs.unlink('./target.txt', err => {
  if (err) {
    console.log(err);
    return;
  }
});4
```

Debugging

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

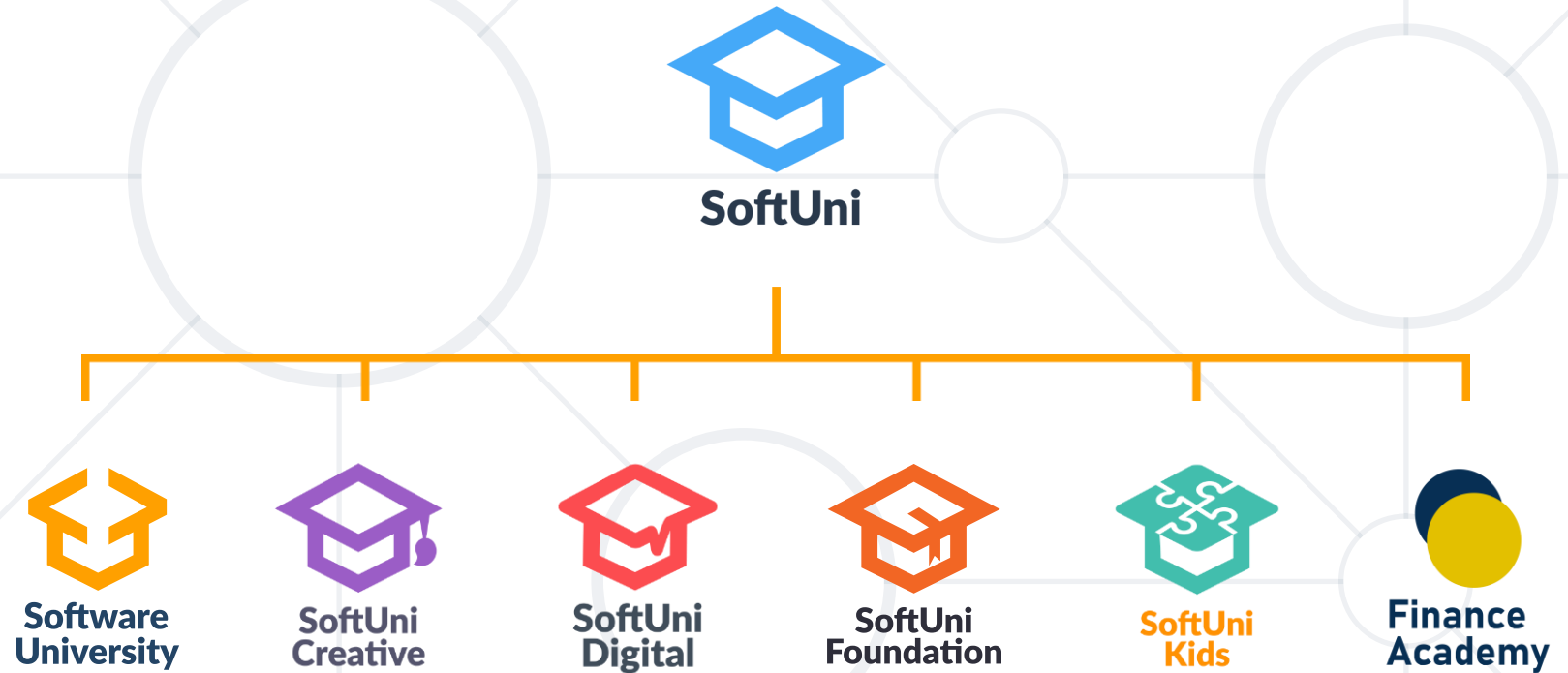 - Node-inspector (not working with latest version)

- 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?

SoftUni Diamond Partners

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg
  - Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**

- Unauthorized copy, reproduction or use is illegal

- © SoftUni – https://about.softuni.bg/

- © Software University – https://softuni.bg