Web Application Development

2020
Node.js, npm, Vue.js
1. Server-side javascript - Node.js
2. Node Structure
3. Node Specifics
4. Modules
5. npm
6. Frameworks
7. Vue.js
**Node.js** is an open-source, cross-platform, JavaScript runtime environment that executes JavaScript code outside of a browser. Node.js lets developers use JavaScript to write command line tools and for server-side scripting—running scripts server-side to produce dynamic web page content before the page is sent to the user's web browser.

Consequently, Node.js represents a "JavaScript everywhere" paradigm,[6] unifying web application development around a single programming language, rather than different languages for server- and client-side scripts.[1]

**Server-side Javascript**

**Pros**
- Node is **Fast**
- **Non-blocking** I/O
- Javascript everywhere
- Large ecosystem (**npm**)  

**Cons**
- Have to think in **async**  
- Callback **Hell**  
- Not as **mature** as other technologies
Multithreaded Server (e.g. Java)
**CPU intensive tasks**

Non-blocking I/O can’t solve CPU problem.

**Workers** are useful for performing CPU-intensive JavaScript operations; do not use them for I/O, since Node.js’s built-in mechanisms for performing operations asynchronously already treat it more efficiently than Worker threads can.
function hell(win) {
  // for listener purpose
  return function() {
    loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
      loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
        loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
          loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
            loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {
              loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
                loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                  loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                    loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                      async.eachSeries(SCRIPTS, function(src, callback) {
                        loadScript(win, BASE_URL+src, callback);
                      }, function() {
                        });
                      });
                    });
                  });
                });
              });
            });
          });
        });
      });
    });
  }
}

http://callbackhell.com/
Node Specifics
The Global Context

In browser it is `window` object

`window.alert()` is the same as `alert()`

In Node it is `global` object

`global.require()` is the same as `require()`
Node Process

- It's an interface to the current Node process
- Accessible from anywhere (via: `global`)
- It's an `EventEmitter`
Environment Variables

Environment variable can be accessed via `process.env` object:

```javascript
{
  SHELL: '/usr/local/bin/bash',
  PATH: '~/.bin/:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin',
  ...
  PWD: '/Users/maciej',
}
```
Process Events

process.on('uncaughtException', (err) => {
    fs.writeFileSync(1, `Caught exception: ${err}
`);
});

setTimeout(() => { console.log('This will still run.');}, 500);

// Intentionally cause an exception, but don't catch it.
nonexistentFunc();

console.log('This will not run.');
Process Events

```javascript
process.on('SIGINT', () => {
    console.log('Received SIGINT. Press Control-D to exit.'
  });

// Using a single function to handle multiple signals
function handle(signal) {
    console.log(`Received ${signal}`
  );
}

process.on('SIGINT', handle);
process.on('SIGTERM', handle)
```
Process Arguments

You can access process arguments via argv object

Launching the Node.js process as:

```
$ node process-args.js one two=three four
```

Inside process-args.js:

```
process.argv.forEach((val, index) => {
  console.log(`$${index}: $${val}`);
});
```

Would generate the output:

```
0: /usr/local/bin/node
1: /Users/mjr/work/node/process-args.js
2: one
3: two=three
4: four
```
Node Modules
const Square = require('./square.js');

const mySquare = new Square(2);

console.log(`The area of mySquare is ${mySquare.area()}`);
module.exports = class Square {

  constructor(width) {
    this.width = width;
  }

  area() {
    return this.width ** 2;
  }
};
Module Patterns

- Simple Object API
- Function Initialization
- Object Constructor
module.exports = {
  increaseWidth: function(width) {
    this.width += width;
  },
  width: 2,
  area: function() {
    return this.width ** 2;
  }
};

const Square = require('./square.js');
Square.increaseWidth(1);
console.log(`The area is ${Square.area()}`);
// The area is 9

const Square = require('./square.js');
Square.increaseWidth(1);
console.log(`The area is ${Square.area()}`);
// The area is 16

Note: Object modules are cached, and they can cause unexpected behaviour if required multiple times.
Function Initialization Pattern

```javascript
module.exports = function Square() {
  return {
    increaseWidth: function(width) {
      this.width += width;
    },
    width: 2,
    area: function() {
      return this.width ** 2;
    }
  }
};
```

```javascript
const Square = require('./square.js');

const mySquare = Square();
mySquare.increaseWidth(1);
console.log(`The area is ${mySquare.area()}`);
// The area is 9
```

```javascript
const Square = require('./square.js');

const mySquare = Square();
mySquare.increaseWidth(1);
console.log(`The area is ${mySquare.area()}`);
// The area is 9
```
```javascript
module.exports = class Square {

    constructor(width) {
        this.width = width;
    }

    area() {
        return this.width ** 2;
    }
};

const Square = require('./square.js');

const mySquare = new Square();

mySquare.increaseWidth(1);

console.log(`The area is ${mySquare.area()}`);

//The area is 9
```

```javascript
const Square = require('./square.js');

const mySquare = new Square();

mySquare.increaseWidth(1);

console.log(`The area is ${mySquare.area()}`);

//The area is 9
```
NPM

Node Package Manager
npm consists of three distinct components:

- the website: https://www.npmjs.com/
- the Command Line Interface (CLI)
- the registry
Use npm to . . .

- Adapt packages of code for your apps, or incorporate packages as they are.
- Download standalone tools you can use right away.
- Run packages without downloading using `npx`.
- Share code with any npm user, anywhere.
- Restrict code to specific developers.
- Create Orgs (organizations) to coordinate package maintenance, coding, and developers.
- Form virtual teams by using Orgs.
- Manage multiple versions of code and code dependencies.
- Update applications easily when underlying code is updated.
- Discover multiple ways to solve the same puzzle.
- Find other developers who are working on similar problems and projects.
A **package** is a file or directory that is described by a **package.json** file. A package must contain a **package.json** file in order to be published to the npm registry.
{ "name": "jquery",
  "title": "jQuery",
  "description": "JavaScript library for DOM operations",
  "version": "4.0.0-pre",
  "main": "dist/jquery.js",
  "homepage": "https://jquery.com",
  "author": { "name": "JS Foundation and other contributors",
              "url": "https://github.com/jquery/jquery/blob/master/AUTHORS.txt" }
  },
  "repository": { "type": "git",
                  "url": "https://github.com/jquery/jquery.git" }
},

"keywords": ["jquery", "javascript", "browser", "library" ]}
NPM CLI tool: npm

$ npm <command> [args]
$ npm install express
// Or
$ npm install grunt --save
$ npm install mymodule --save-dev

```json
{
  ...
  dependencies: {
    "grunt": "1.0.4"
  },
  devDependencies: {
    "mymodule": "0.0.1"
  },
  ...
}
```
Frameworks
In computer programming, a **software framework** is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software. It provides a standard way to build and deploy applications and it is a universal, reusable **software environment** that provides particular functionality as part of a larger **software platform** to facilitate development of **software applications**, products and solutions. Software frameworks may include support programs, compilers, code libraries, tool sets, and **application programming interfaces (APIs)** that bring together all the different **components** to enable development of a **project** or **system**.

Why use frameworks

- If your application gets complicated
- If you need to handle a lot of data and/or UI elements
- Framework code is better tested and more reliable
- It forces you to use proper code style
Vue.js
Vue.js

Main concepts

- Progressive
- Modular
- Reactive
Components
Why components are a good idea

```
<div class="controls">
  <button id="profile-button" class="pill active">Profile</button>
  <button id="courses-button" class="pill">Courses</button>
</div>
```
<template>
  <p>{{ greeting }} World!</p>
</template>

<script>
module.exports = {
  data: function () {
    return {
      greeting: 'Hello'
    }
  }
}
</script>

<style scoped>
  p {
    font-size: 2em;
    text-align: center;
  }
</style>
References

https://nodejs.org/en/

https://www.npmjs.com/

https://vuejs.org/
Questions?

Next: Vue.js