Node.js Fundamentals

Dave van Soest

January 13, 2015

Technologies

- V8 (Javascript engine created by Google)

Technologies

- V8 (Javascript engine created by Google)
- C++ (Libraries)

Asynchronous programming

Single-threaded process

The Node application is executed as a single-threaded process.

Asynchronous programming

Single-threaded process

The Node application is executed as a single-threaded process.

Consequences

Asynchronous programming

Single-threaded process

The Node application is executed as a single-threaded process.

Consequences

- The application can only be busy doing one thing at a time.

Asynchronous programming

Single-threaded process

The Node application is executed as a single-threaded process.

Consequences

- The application can only be busy doing one thing at a time.
- Long-running calculations will block other work.

Asynchronous programming

Single-threaded process

The Node application is executed as a single-threaded process.

Consequences

- The application can only be busy doing one thing at a time.
- Long-running calculations will block other work.
- Blocking system calls will pause the application.

Asynchronous programming

Non-blocking I/O

Asynchronous programming

Non-blocking I/O

- Operations using peripheral hardware (disks, network interfaces, timers, ...) take a relatively long time to complete.

Asynchronous programming

Non-blocking I/O

- Operations using peripheral hardware (disks, network interfaces, timers, ...) take a relatively long time to complete.
- Non-blocking system calls return immediately, leaving the result of the operation to be fetched with a later system call.

Asynchronous programming

Non-blocking I/O

- Operations using peripheral hardware (disks, network interfaces, timers, ...) take a relatively long time to complete.
- Non-blocking system calls return immediately, leaving the result of the operation to be fetched with a later system call.

Simplified example

Asynchronous programming

Non-blocking I/O

- Operations using peripheral hardware (disks, network interfaces, timers, ...) take a relatively long time to complete.
- Non-blocking system calls return immediately, leaving the result of the operation to be fetched with a later system call.

Simplified example

- Start reading a block of data from a file descriptor (system data resource) using non-blocking I/O.

Asynchronous programming

Non-blocking I/O

- Operations using peripheral hardware (disks, network interfaces, timers, ...) take a relatively long time to complete.
- Non-blocking system calls return immediately, leaving the result of the operation to be fetched with a later system call.

Simplified example

- Start reading a block of data from a file descriptor (system data resource) using non-blocking I/O.
- Loop:

Asynchronous programming

Non-blocking I/O

- Operations using peripheral hardware (disks, network interfaces, timers, ...) take a relatively long time to complete.
- Non-blocking system calls return immediately, leaving the result of the operation to be fetched with a later system call.

Simplified example

- Start reading a block of data from a file descriptor (system data resource) using non-blocking I/O.
- Loop:
    - Check whether read operation is completed.

Asynchronous programming

Non-blocking I/O

- Operations using peripheral hardware (disks, network interfaces, timers, ...) take a relatively long time to complete.
- Non-blocking system calls return immediately, leaving the result of the operation to be fetched with a later system call.

Simplified example

- Start reading a block of data from a file descriptor (system data resource) using non-blocking I/O.
- Loop:
    - Check whether read operation is completed.
    - If read data available:

Asynchronous programming

Non-blocking I/O

- Operations using peripheral hardware (disks, network interfaces, timers, ...) take a relatively long time to complete.
- Non-blocking system calls return immediately, leaving the result of the operation to be fetched with a later system call.

Simplified example

- Start reading a block of data from a file descriptor (system data resource) using non-blocking I/O.
- Loop:
    - Check whether read operation is completed.
    - If read data available:
        - Do something with the data.

Asynchronous programming

Non-blocking I/O

- Operations using peripheral hardware (disks, network interfaces, timers, ...) take a relatively long time to complete.
- Non-blocking system calls return immediately, leaving the result of the operation to be fetched with a later system call.

Simplified example

- Start reading a block of data from a file descriptor (system data resource) using non-blocking I/O.
- Loop:
    - Check whether read operation is completed.
    - If read data available:
        - Do something with the data.
    - If read error:

Asynchronous programming

Non-blocking I/O

- Operations using peripheral hardware (disks, network interfaces, timers, ...) take a relatively long time to complete.
- Non-blocking system calls return immediately, leaving the result of the operation to be fetched with a later system call.

Simplified example

- Start reading a block of data from a file descriptor (system data resource) using non-blocking I/O.
- Loop:
    - Check whether read operation is completed.
    - If read data available:
        - Do something with the data.
    - If read error:
        - Display error message.

Asynchronous programming

Non-blocking I/O: Multiple resources

Example: Multiple data resources

Asynchronous programming

Non-blocking I/O: Multiple resources

Example: Multiple data resources

- Listen for incoming network connections (non-blocking).

Asynchronous programming

Non-blocking I/O: Multiple resources

Example: Multiple data resources

- Listen for incoming network connections (non-blocking).
- Loop:

Asynchronous programming

Non-blocking I/O: Multiple resources

Example: Multiple data resources

- Listen for incoming network connections (non-blocking).
- Loop:
    - Check for events.

Asynchronous programming

Non-blocking I/O: Multiple resources

Example: Multiple data resources

- Listen for incoming network connections (non-blocking).
- Loop:
  - Check for events.
  - If new incoming connection:

Asynchronous programming

Non-blocking I/O: Multiple resources

Example: Multiple data resources

- Listen for incoming network connections (non-blocking).
- Loop:
    - Check for events.
    - If new incoming connection:
        - Start receiving data from incoming connection (non-blocking).

Asynchronous programming

Non-blocking I/O: Multiple resources

Example: Multiple data resources

- Listen for incoming network connections (non-blocking).
- Loop:
  - Check for events.
  - If new incoming connection:
    - Start receiving data from incoming connection (non-blocking).
  - If data received from connection:

Asynchronous programming

Non-blocking I/O: Multiple resources

Example: Multiple data resources

- Listen for incoming network connections (non-blocking).
- Loop:
    - Check for events.
    - If new incoming connection:
        - Start receiving data from incoming connection (non-blocking).
    - If data received from connection:
        - Process data and generate response.

Asynchronous programming

Non-blocking I/O: Multiple resources

Example: Multiple data resources

- Listen for incoming network connections (non-blocking).
- Loop:
  - Check for events.
  - If new incoming connection:
    - Start receiving data from incoming connection (non-blocking).
  - If data received from connection:
    - Process data and generate response.
    - Start sending response data to connection (non-blocking).

Asynchronous programming

Non-blocking I/O: Multiple resources

Example: Multiple data resources

- Listen for incoming network connections (non-blocking).
- Loop:
    - Check for events.
    - If new incoming connection:
        - Start receiving data from incoming connection (non-blocking).
    - If data received from connection:
        - Process data and generate response.
        - Start sending response data to connection (non-blocking).
    - If response data is sent:

Asynchronous programming

Non-blocking I/O: Multiple resources

Example: Multiple data resources

- Listen for incoming network connections (non-blocking).
- Loop:
  - Check for events.
  - If new incoming connection:
    - Start receiving data from incoming connection (non-blocking).
  - If data received from connection:
    - Process data and generate response.
    - Start sending response data to connection (non-blocking).
  - If response data is sent:
    - Close the connection.

Asynchronous programming

Non-blocking I/O: Multiple resources

Example: Multiple data resources

- Listen for incoming network connections (non-blocking).
- Loop:
  - Check for events.
  - If new incoming connection:
    - Start receiving data from incoming connection (non-blocking).
  - If data received from connection:
    - Process data and generate response.
    - Start sending response data to connection (non-blocking).
  - If response data is sent:
    - Close the connection.

Callback functions

Asynchronous programming

Non-blocking I/O: Multiple resources

Example: Multiple data resources

- Listen for incoming network connections (non-blocking).
- Loop:
    - Check for events.
    - If new incoming connection:
        - Start receiving data from incoming connection (non-blocking).
    - If data received from connection:
        - Process data and generate response.
        - Start sending response data to connection (non-blocking).
    - If response data is sent:
        - Close the connection.

Callback functions

For each combination of *[resource, event–type]* the application registers a callback function to be executed when the event occurs.

Asynchronous programming

Non-blocking I/O: Optimizing performance

Optimizing performance

Asynchronous programming

Non-blocking I/O: Optimizing performance

Optimizing performance

- Continuous checking (a.k.a. polling) by the application would use a lot of system resources.

Asynchronous programming

Non-blocking I/O: Optimizing performance

Optimizing performance

- Continuous checking (a.k.a. polling) by the application would use a lot of system resources.
- Solution: The *select*–family of system calls (blocking).

Asynchronous programming

Non-blocking I/O: Optimizing performance

Optimizing performance

- Continuous checking (a.k.a. polling) by the application would use a lot of system resources.
- Solution: The *select*–family of system calls (blocking).
    - Monitor multiple system resources for events at the same time.

Asynchronous programming

Non-blocking I/O: Optimizing performance

Optimizing performance

- Continuous checking (a.k.a. polling) by the application would use a lot of system resources.
- Solution: The *select*–family of system calls (blocking).
  - Monitor multiple system resources for events at the same time.
  - Return when an event occurs, a timer expires or a signal is received.

Asynchronous programming

Non-blocking I/O: Optimizing performance

Optimizing performance

- Continuous checking (a.k.a. polling) by the application would use a lot of system resources.
- Solution: The *select*–family of system calls (blocking).
    - Monitor multiple system resources for events at the same time.
    - Return when an event occurs, a timer expires or a signal is received.
    - Uses minimal amount of system resources. Application itself is idle.

Asynchronous programming

Non-blocking I/O: Optimizing performance

Optimizing performance

- Continuous checking (a.k.a. polling) by the application would use a lot of system resources.
- Solution: The *select*–family of system calls (blocking).
  - Monitor multiple system resources for events at the same time.
  - Return when an event occurs, a timer expires or a signal is received.
  - Uses minimal amount of system resources. Application itself is idle.
- Leverages:

Asynchronous programming

Non-blocking I/O: Optimizing performance

Optimizing performance

- Continuous checking (a.k.a. polling) by the application would use a lot of system resources.
- Solution: The *select*–family of system calls (blocking).
  - Monitor multiple system resources for events at the same time.
  - Return when an event occurs, a timer expires or a signal is received.
  - Uses minimal amount of system resources. Application itself is idle.
- Leverages:
  - Software interrupts

Asynchronous programming

Non-blocking I/O: Optimizing performance

Optimizing performance

- Continuous checking (a.k.a. polling) by the application would use a lot of system resources.
- Solution: The *select*–family of system calls (blocking).
    - Monitor multiple system resources for events at the same time.
    - Return when an event occurs, a timer expires or a signal is received.
    - Uses minimal amount of system resources. Application itself is idle.
- Leverages:
    - Software interrupts
    - Hardware interrupts

Asynchronous programming

Non-blocking I/O: Optimizing performance

Optimizing performance

- Continuous checking (a.k.a. polling) by the application would use a lot of system resources.
- Solution: The *select*–family of system calls (blocking).
  - Monitor multiple system resources for events at the same time.
  - Return when an event occurs, a timer expires or a signal is received.
  - Uses minimal amount of system resources. Application itself is idle.
- Leverages:
  - Software interrupts
  - Hardware interrupts
  - Direct Memory Access (D.M.A.)

Asynchronous programming

Advantages

Advantages

Asynchronous programming

Advantages

Advantages

- Optimizing system resource usage:

Asynchronous programming

Advantages

Advantages

- Optimizing system resource usage:
    - Running operations on multiple data resources in parallel instead of sequential.

Asynchronous programming

Advantages

Advantages

- Optimizing system resource usage:
    - Running operations on multiple data resources in parallel instead of sequential.
    - Reduces relatively expensive process-switching and thread-switching.

Asynchronous programming

Advantages

Advantages

- Optimizing system resource usage:
    - Running operations on multiple data resources in parallel instead of sequential.
    - Reduces relatively expensive process-switching and thread-switching.
- No worries about thread-safety.

Asynchronous programming

Javascript

Javascript is perfectly suited for asynchronous programming.

Asynchronous programming

Javascript

Javascript is perfectly suited for asynchronous programming.

- Javascript engines have a built–in event–loop.

Asynchronous programming

Javascript

Javascript is perfectly suited for asynchronous programming.

- Javascript engines have a built–in event–loop.
- First-class functions are used as callbacks.

Asynchronous programming

Javascript

Javascript is perfectly suited for asynchronous programming.

- Javascript engines have a built–in event–loop.
- First-class functions are used as callbacks.
- Anonymous functions allow for in-line callbacks.

Asynchronous programming

Javascript

Javascript is perfectly suited for asynchronous programming.

- Javascript engines have a built–in event–loop.
- First-class functions are used as callbacks.
- Anonymous functions allow for in-line callbacks.
- Automatic binding of a function to its outside context (closure) allows for easy access of program state inside a callback.

Simple example

HTTP-server example from Node.js homepage

```
var http = require("http");

var httpServer = http.createServer(
    function (req, res) {
        res.writeHead(200, {"Content-Type": "text/plain"});
        res.end("Hello␣World\n");
    }
);
httpServer.listen(1337, "127.0.0.1");

console.log("Server␣running␣at␣http://127.0.0.1:1337/");
```

Extended example

Simple file server

```
var http = require("http");
var fs = require("fs");

var httpServer = http.createServer(
    function (req, res) {
        fs.readFile(req.url, function(err, data) {
            res.writeHead(err ? 404 : 200, {"Content-Type": "text/plain"});
            res.end(err ? "Not␣found.\n" : data);
        });
    }
);
httpServer.listen(1337, "127.0.0.1");
```

Example using streams and event emitters

Streaming file server

```
var http = require("http");
var fs = require("fs");

var httpServer = http.createServer(
    function (req, res) {
        var stream = fs.createReadStream(req.url);

        stream.on("open", function() {
            res.writeHead(200, {"Content-Type": "text/plain"});
            stream.pipe(res);
        });

        stream.on("error", function(err) {
            res.writeHead(400, {"Content-Type": "text/plain"});
            res.end(err);
        });
    }
);
httpServer.listen(1337, "127.0.0.1");
```

API

Selection from the Node.js built-in API:

API

Selection from the Node.js built-in API:

- http: HTTP-server and client.

API

Selection from the Node.js built-in API:

- http: HTTP-server and client.
- net: TCP server and client.

API

Selection from the Node.js built-in API:

- http: HTTP-server and client.
- net: TCP server and client.
- fs: Filesystem operations (reading, writing, listing, ...).

API

Selection from the Node.js built-in API:

- http: HTTP-server and client.
- net: TCP server and client.
- fs: Filesystem operations (reading, writing, listing, ...).
    - Both non-blocking and blocking operations!

API

Selection from the Node.js built-in API:

- http: HTTP-server and client.
- net: TCP server and client.
- fs: Filesystem operations (reading, writing, listing, ...).
    - Both non-blocking and blocking operations!
- path: File path handling.

API

Selection from the Node.js built-in API:

- http: HTTP-server and client.
- net: TCP server and client.
- fs: Filesystem operations (reading, writing, listing, ...).
  - Both non-blocking and blocking operations!
- path: File path handling.
- readline: Create a text-based user interface.

API

Selection from the Node.js built-in API:

- http: HTTP-server and client.
- net: TCP server and client.
- fs: Filesystem operations (reading, writing, listing, ...).
  - Both non-blocking and blocking operations!
- path: File path handling.
- readline: Create a text-based user interface.
- url, querystring: URL parsing and generating.

API

Selection from the Node.js built-in API:

- http: HTTP-server and client.
- net: TCP server and client.
- fs: Filesystem operations (reading, writing, listing, ...).
  - Both non-blocking and blocking operations!
- path: File path handling.
- readline: Create a text-based user interface.
- url, querystring: URL parsing and generating.
- globals: global, process, console, module, ...

API

Selection from the Node.js built-in API:

- http: HTTP-server and client.
- net: TCP server and client.
- fs: Filesystem operations (reading, writing, listing, ...).
  - Both non-blocking and blocking operations!
- path: File path handling.
- readline: Create a text-based user interface.
- url, querystring: URL parsing and generating.
- globals: global, process, console, module, ...
- ...

Modules

Using a module

Using a module

Modules

Using a module

Using a module

- Importing a module:
  `var xyz = require("xyz");`
  (short-hand for `module.require`)

Modules

Using a module

Using a module

- Importing a module:
  `var xyz = require("xyz");`
  (short-hand for `module.require`)
  - Module is searched for in local `node_modules` directory and in the system's modules directory.

Modules

Using a module

Using a module

- Importing a module:
  `var xyz = require("xyz");`
  (short-hand for `module.require`)
    - Module is searched for in local `node_modules` directory and in the system's modules directory.
- Importing a local module:
  `var abc = require("./abc");`

Modules

Using a module

Using a module

- Importing a module:
  `var xyz = require("xyz");`
  (short-hand for `module.require`)
  - Module is searched for in local `node_modules` directory and in the system's modules directory.
- Importing a local module:
  `var abc = require("./abc");`
  - Module is searched for in local directory.

Modules

Module initialization

Example module

```
// Initialization code:
console.log("Hello␣ABC.");

// Exporting functionality:
exports.doAbc = function() {
    console.log("Performing␣ABC.");
};
```

Modules

Module initialization

Example module

```
// Initialization code:
console.log("Hello ABC.");

// Exporting functionality:
exports.doAbc = function() {
    console.log("Performing ABC.");
};
```

Module initialization

Modules

Module initialization

Example module

```
// Initialization code:
console.log("Hello␣ABC.");

// Exporting functionality:
exports.doAbc = function() {
    console.log("Performing␣ABC.");
};
```

Module initialization

- A module is initialized upon the first import.

Modules

Module initialization

Example module

```
// Initialization code:
console.log("Hello␣ABC.");

// Exporting functionality:
exports.doAbc = function() {
    console.log("Performing␣ABC.");
};
```

Module initialization

- A module is initialized upon the first import.
- Each module is only initialized once within the program.

Modules

Creating a module

Creating a module

Modules

Creating a module

Creating a module

- A local module can be ...

Modules

Creating a module

Creating a module

- A local module can be ...
  - a single file `abc.js` or a directory

Modules

Creating a module

Creating a module

- A local module can be ...
    - a single file `abc.js` or a directory
    - a directory `abc` with a file called `index.js` as the module's main entry point

Modules

Creating a module

Creating a module

- A local module can be ...
  - a single file `abc.js` or a directory
  - a directory `abc` with a file called `index.js` as the module's main entry point
  - a directory `abc` with a package definition file `package.json`

Modules

Creating a module

Creating a module

- A local module can be ...
  - a single file `abc.js` or a directory
  - a directory `abc` with a file called `index.js` as the module's main entry point
  - a directory `abc` with a package definition file `package.json`
    - This type of module is called a package.

Modules

Creating a module

Creating a module

- A local module can be ...
    - a single file `abc.js` or a directory
    - a directory `abc` with a file called `index.js` as the module's main entry point
    - a directory `abc` with a package definition file `package.json`
        - This type of module is called a package.
- A module can import other modules.

Modules

Creating a module

Creating a module

- A local module can be ...
  - a single file `abc.js` or a directory
  - a directory `abc` with a file called `index.js` as the module's main entry point
  - a directory `abc` with a package definition file `package.json`
    - This type of module is called a package.
- A module can import other modules.
- A module exports data and functionality through the `exports` object (short-hand for `module.exports`)

Modules

Local module usage example

./abc.js

```
// Initialization code:
console.log("Hello␣ABC.");

// Exporting functionality:
exports.doAbc = function() {
    console.log("Performing␣ABC.");
};
```

Modules

Local module usage example

./abc.js

```
// Initialization code:
console.log("Hello␣ABC.");

// Exporting functionality:
exports.doAbc = function() {
    console.log("Performing␣ABC.");
};
```

./app.js

```
var abc = require("./abc");

abc.doAbc();
```

Packages

Package definition

File: `package.json` in module directory

```
{
    "name": "abc",
    "version": "1.0.0",
    "description": "A,␣B␣and␣C",
    "main": "index.js",
    "dependencies": {
        "underscore": "~1.6.0"
        "xyz": "1.2.3"
    }
    ...
}
```

Packages

Node Package Manager (NPM)

Packages

Node Package Manager (NPM)

- Installing packages locally (in `node_modules` directory) or globally.

Packages

Node Package Manager (NPM)

- Installing packages locally (in `node_modules` directory) or globally.
- Updating packages.

Packages

Node Package Manager (NPM)

- Installing packages locally (in `node_modules` directory) or globally.
- Updating packages.
- Package version management.

Packages

Node Package Manager (NPM)

- Installing packages locally (in `node_modules` directory) or globally.
- Updating packages.
- Package version management.
- Publishing packages.